Introducción a los

SISTEMAS DE BASES DE DATOS

Prentice Hall

C J . A T E

V

V#	PROVEEDOR	ESTADO	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

VP

v#	P#	CANT
V1	P1	300
V1	P2	200
V1	P3	400
V1	P4	200
V1	P5	100
V1	P6	100
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200
V4	P4	300
V4	P5	400

p

P#	PARTE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	12	Londres
P2	Perno	Verde	17	París
P3	Tornillo	Azul	17	Roma
P4	Tornillo	Rojo	14	Londres
P5	Leva	Azul	12	París
P6	Engrane	Rojo	19	Londres

La base de datos de proveedores y partes (valores de ejemplo)

V

V#	PROVEEDOR	ESTADO	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

VPY

V#	P#	Y#	CANT
V1	P1	Y1	200
V1	P1	Y4	700
V2	P3	Y1	400
V2	P3	Y2	200
V2	P3	Y3	200
V2	P3	Y4	500
V2	P3	Y5	600
V2	P3	Y6	400
V2	P3	Y7	600
V2	P5	Y2	100
V3	P3	Y1	200
V3	P4	Y2	500
V4	P6	Y3	300
V4	P6	Y7	300
V5	P2	Y2	200
V5	P2	Y4	100
V5	P5	Y5	500
V5	P5	Y7	100
V5	P6	Y2	200
V5	P1	Y4	100
V5	P3	Y4	200
V5	P4	Y4	800
V5	P5	Y4	400
V5	P6	Y4	500

p

p#	PARTE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	12.0	Londres
P2	Perno	Verde	17.0	París
P3	Tornillo	Azul	17.0	Roma
P4	Tornillo	Rojo	14.0	Londres
P5	Leva	Azul	12.0	París
P6	Engrane	Rojo	19.0	Londres

Y

Y#	PROYECTO	CIUDAD
Y1	Clasificador	París
Y2	Monitor	Roma
Y3	OCR	Atenas
Y4	Consola	Atenas
Y5	RAID	Londres
Y6	EDS	Oslo
Y7	Cinta	Londres

La base de datos proveedores, partes y proyectos (valores de ejemplo)

INTRODUCCIÓN A LOS Sistemas de bases de datos

SÉPTIMA EDICIÓN

C. J. Date

TRADUCCIÓN:
I. Q. Sergio Luis María Ruiz Faudón
Ingeniero Químico, Analista de Sistemas
Sergio Kourchenko Barrena

REVISIÓN TÉCNICA:
Dr. Felipe López Gamino
Instituto Tecnológico Autónomo de México

Pearson Educación ®

MÉXICO • ARGENTINA • BRASIL • COLOMBIA • COSTA RICA • CHILE
ESPAÑA • GUATEMALA • PERÚ • PUERTO RICO • VENEZUELA

Datos de catalogación bibliográfica

DATE, C. J.
Introducción a **los** sistemas **de** bases de datos

PEARSON EDUCACIÓN, México, 2001

ISBN: 968-444-419-2
Área: Universitarios

Formato: 18.5 x 23.5 cm Páginas: 960

Versión en español de la obra titulada An *introduction to datábase systems. Seventh Edition,* de C. J. Date, publicada originalmente en inglés por Addison Wesley Longman, Inc., Reading Massachusetts. E.U.A.

Esta edición en español es la única autorizada.

Original English language title by
Addison Wesley Longman, Inc.
 ISBN 0-201-38590-2

Edición en español
Editor: José Luis Vázquez
Supervisor de traducción: Antonio Núñez Ramos
Supervisor de producción: Enrique Trejo Hernández

Edición en inglés:
Acquisitions Editor: Maite Suarez-Rivas
Associate Editor: Katherine Harutunian
Production Services: Solar Script, Inc.
Composition: Publishers' Design and Production Services, Inc.
Cover Design: Night & Day Design

SÉPTIMA EDICIÓN, 2001

Atlacomulco Núm. 500-5° Piso Col. Industrial Atoto
53519, Naucalpan de Juárez, Edo. de México

Cámara Nacional de la Industria Editorial Mexicana
Reg. Núm. 1031

Pearson Educación ®

ISBN 968-444-419-2
Impreso en México. *Printed in México*

1 2 3 4 5 6 7 8 9 0 - 03 02 01 00

Este libro está dedicado a mi esposa Lindy
y ala memoria de mi madre Rene

Ac<

Acerca del autor

C.J. Date es autor, conferencista, investigador y consultor independiente, especializado en la tecnología de bases de datos. Reside en Healdsburg, California.

En 1967, después de varios años como programador matemático e instructor de programación en Leo Computers Ltd. (Londres, Inglaterra), el señor Date se cambió a los Laboratorios de Desarrollo de IBM (Reino Unido), donde trabajó en la integración de la funcionalidad de base de datos dentro de PL/I. En 1974 se trasladó al Centro de Desarrollo de Sistemas de IBM en California, donde fue responsable del diseño de un lenguaje de base de datos conocido como UDL (Lenguaje Unificado de Bases de Datos) y trabajó en la planeación técnica, así como en el diseño externo de los productos SQL/DS y DB2 de IBM. Dejó IBM en mayo de 1983.

El señor Date ha estado activo en el campo de las bases de datos por más de 30 años. Fue uno de los primeros en reconocer la importancia del trabajo pionero de Codd sobre el modelo relational. Ha dado muchas conferencias sobre aspectos técnicos —principalmente sobre temas de bases de datos y, en especial, sobre bases de datos relaciónales— en todo el territorio norteamericano y también en Europa, Australia, América Latina y el Lejano Oriente. Además de este libro, es autor o coautor de varios otros libros sobre bases de datos, incluyendo *Foundation for Object/Relational Databases: The Third Manifesto* (1998), una propuesta detallada de la dirección futura del campo; *Database: A Primer* (1983), que aborda los sistemas de bases de datos desde el punto de vista de un no especialista; una serie de libros: *Relational Database Writings* (1986, 1990, 1992, 1995, 1998), que tratan a profundidad los diversos aspectos de la tecnología relational; y otra serie de libros sobre sistemas y lenguajes en particular: *A Guide to DB2* (cuarta edición, 1993), *A Guide to SYBASE and SQL Server* (1992), *A Guide to SQL/DS* (1988), *A Guide to INGRES* (1987) y *A Guide to the SQL Standard* (cuarta edición, 1997). Sus libros se han traducido a muchos idiomas, incluyendo chino, holandés, francés, alemán, griego, italiano, japonés, coreano, polaco, portugués, ruso, español y Braille.

El señor Date también ha producido más de 300 artículos y documentos de investigación y ha hecho diversas contribuciones originales a la teoría de las bases de datos. Es columnista regular de las revistas *Database Programming & Design* e *Intelligent Enterprise.* Sus seminarios profesionales sobre tecnología de bases de datos (que ofrece tanto en Norteamérica como en el extranjero), se consideran ampliamente como los más importantes por la calidad de los temas y la claridad en la exposición.

El señor Date posee un grado de Honor en Matemáticas de la Universidad de Cambridge, Inglaterra (BA 1962, MA 1966) y el grado honorario de Doctor de Tecnología de la Universidad de Montfort, Inglaterra (1994).

Co
1

CAPÍTUI

CAPITUl

Contenido

APÉNDICES 887

Prefacio a la séptima edición

Este libro es una amplia introducción al ahora muy extendido campo de los sistemas de bases de datos. Proporciona una base sólida en los fundamentos de las tecnologías de bases de datos y ofrece cierta idea sobre cómo podría desarrollarse este campo en el futuro. El libro está concebido principalmente como un libro de texto, no como una referencia de trabajo (aunque creo que también puede ser útil, hasta cierto grado, como referencia); a todo lo largo del mismo, se hace énfasis en la **profundidad y** en la **comprensión,** no sólo en los formalismos.

PRERREQUISITOS

El libro en su conjunto está destinado para todo aquel que esté de alguna forma interesado profesionalmente y que desee comprender de qué se tratan los sistemas de bases de datos. Doy por hecho que usted tiene por lo menos un conocimiento básico de:

- Las posibilidades de almacenamiento y administración de archivos (indexación, etcétera) de un sistema moderno de computadora;
- Las características de uno o más lenguajes de programación de alto nivel (por ejemplo, C, Java, Pascal, PL/I, etcétera).

ESTRUCTURA

El libro está dividido en seis partes principales:

I. Preliminares
II. El modelo relacional
III. Diseño de bases de datos
IV. Administración de transacciones
V. Temas adicionales
VI. Bases de datos de objetos y de objetos/relaciónales

Cada parte está dividida a su vez en diversos capítulos:

- La parte I (cuatro capítulos) ofrece una amplia introducción al concepto de los sistemas de bases de datos en general y a los sistemas relaciónales en particular. También presenta el lenguaje estándar de base de datos, **SQL.**
- La parte II (cinco capítulos) consiste en una descripción detallada y muy cuidadosa del **modelo** relacional, que no solamente es el fundamento teórico subyacente de los sistemas relaciónales sino que de hecho es el fundamento teórico del campo de las bases de datos en su conjunto.
- La parte III (cuatro capítulos) expone la cuestión general del **diseño de bases de datos;** tres capítulos están dedicados a la teoría del diseño, el cuarto considera el modelado de la semántica y el modelo entidad/vínculo.
- La parte IV (dos capítulos) se refiere a la **administración de transacciones** (es decir, los controles de recuperación y concurrencia).
- La parte V (ocho capítulos) es un poco como un *popurrí.* Sin embargo, muestra en general que los conceptos relacionales son relevantes para una variedad de aspectos adicionales de la tecnología de bases de datos —s**eguridad, bases de datos distribuidas, datos tempo rales, apoyo a la toma de decisiones,** etcétera.
- Por último, la parte VI (dos capítulos) describe el impacto de la **tecnología de objetos** en **los** sistemas de bases de datos. En particular, el capítulo 25 —el último del libro— consi dera la posibilidad de una *aproximación* entre las tecnologías de objetos y relacional y ex pone sistemas de **objetos/relaciónales.**

También hay tres apéndices: uno que ofrece detalles adicionales de SQL, otro sobre "SQL3" (una nueva versión de SQL que probablemente para cuando este libro esté impreso, ya haya sido ratificada como estándar) y un tercero que presenta una lista de abreviaturas y acrónimos importantes.

Nota: También está disponible un *manual en línea del instructor* (en inglés), que ofrece una guía sobre cómo utilizar este libro como base de un curso de enseñanza sobre bases de datos. Consta de una serie de notas, consejos y sugerencias sobre cada parte, capítulo y apéndice, así como las respuestas a los ejercicios que no están incluidas en el propio libro y otros materiales de apoyo. Para instrucciones sobre cómo acceder al *manual,* envíe un mensaje de correo electrónico a la siguiente dirección: luis.vazquez@pearsoned.com.

COMO LEER ESTE LIBRO

En general, el libro está destinado para ser leído básicamente siguiendo la secuencia en la que fue escrito, aunque si usted lo prefiere, puede saltarse los últimos capítulos o las últimas secciones dentro de los capítulos. Un plan sugerido para una primera lectura sería:

- Leer los capítulos 1 y 2 "a la ligera";
- Leer los capítulos 3 y 4 muy detenidamente;
- Leer los capítulos 5, 6, 8 y 9 con detenimiento, pero saltando el capítulo 7 (excepto la sec ción 7.7);
- Leer el capítulo 10 "a la ligera";
- Leer los capítulos 11 y 13 detenidamente, saltando el capítulo 12;
- Leer los capítulos 14 y 15 detenidamente;
- Leer los capítulos subsiguientes de manera selectiva, de acuerdo con su gusto e intereses.

Cada capítulo inicia con una introducción y termina con un resumen; además, la mayoría de los capítulos incluyen un conjunto de ejercicios, con sus respectivas respuestas en la mayoría de los casos (a menudo las respuestas dan información adicional sobre el tema del ejercicio). La mayoría de los capítulos también incluyen una amplia lista de referencias bibliográficas, muchas de ellas comentadas. Esta estructura permite tratar los temas por importancia, presentando los conceptos y resultados más importantes dentro del cuerpo principal del texto y dejando varios puntos suplementarios y aspectos más complejos para las secciones de ejercicios, respuestas o referencias, según corresponda. *Nota:* Las referencias se identifican mediante dos números entre corchetes. Por ejemplo, la referencia "[3.1]" indica el primer elemento de la lista de referencias al final del capítulo 3; o sea, un documento de E. F. Codd publicado en *CACM,* Vol. 25 Núm. 2, en febrero de 1982 (consulte el apéndice C para una explicación de las abreviaturas —como CACM— empleadas en las referencias).

COMPARACIÓN CON EDICIONES ANTERIORES

A continuación describimos las diferencias principales entre esta edición y su predecesora inmediata.

- *Parte I:* Los capítulos 1 a 3 cubren más o menos el mismo terreno que sus correspondientes en la edición anterior, aunque fueron reescritos mejorando y ampliando el tratamiento de va rios temas. El capítulo 4 es nuevo (aunque está basado parcialmente en el anterior capítulo 8); y ofrece una introducción a SQL, abordando aspectos que no pertenecen de manera ló gica a ninguna otra parte del libro (en particular, los enlaces del lenguaje anfitrión y el SQL incrustado).
- *Parte II:* Los capítulos 5 al 9 (sobre el modelo relacional) representan una versión reescrita, considerablemente ampliada y muy mejorada de los capítulos 4 al 7 y 17 de la edición an terior. En particular, las secciones sobre tipos (dominios), valores contra variables de relación, integridad, predicados y vistas fueron revisadas de manera drástica.

  Nota: Cabe aquí un comentario. Las ediciones anteriores utilizaban SQL para ilustrar ideas relaciónales, en la creencia de que es más fácil mostrar al estudiante lo concreto antes que lo abstracto. Sin embargo, por desgracia la brecha entre SQL y el modelo relacional ha crecido tanto que ahora creo que sería francamente engañoso usar SQL para dicho fin. De hecho, SQL en su forma actual está muy lejos de ser una verdadera representación de los principios relaciónales —presenta demasiadas faltas tanto de acción como de omisión— que habría preferido relegarlo a un apéndice; pero el lenguaje es tan importante desde el punto de vista comercial (y todo profesional de base de datos necesita tener una cierta familiaridad con él) que no sería apropiado tratarlo de manera tan discriminatoria. Por lo tanto, establecí un acuerdo: un capítulo en la parte I del libro y en secciones de otros capítulos (según corresponda) que describen aquellos aspectos de SQL que son específicos para el tema del capítulo en cuestión.
- *Parte III:* Los capítulos 10 al 13 son una importante revisión de los capítulos 9 al 12 ante riores, con material nuevo sobre los atributos valuados por relación, la desnormalización, el diseño ortogonal y los enfoques alternos al modelado semántico (incluyendo las "reglas del negocio").

  Nota: Una vez más, vale la pena profundizar un poco aquí. Algunos revisores de ediciones anteriores se quejaron de que los aspectos de diseño de base de datos fueron trata-

dos ya avanzado el libro. Pero mi idea personal es que los estudiantes no están listos para diseñar bases de datos en forma adecuada ni para apreciar totalmente los aspectos de diseño hasta que hayan entendido lo que son las bases de datos y cómo se usan; en otras palabras, creo que es importante dedicar algún tiempo al modelo relacional y aspectos afines, antes de exponer al estudiante a las cuestiones de diseño. Por lo tanto, sigo creyendo que la parte **III** está en la posición correcta dentro del libro.

- *Parte IV:* Los dos capítulos de esta parte son versiones ligeramente revisadas y ampliadas de los capítulos 13 y 14 de la edición anterior.
- *Parte V:* Los capítulos 19 (sobre la herencia de tipos), 21 (sobre el apoyo a la toma de de cisiones) y 22 (sobre las bases de datos temporales), son completamente nuevos. Los capí tulos 16 (sobre la seguridad), 17 (sobre la optimización), 18 (sobre la información faltante) y 20 (sobre las bases de datos distribuidas) son versiones revisadas y ampliadas de manera importante de los capítulos anteriores 15, 18, 20 y 21, respectivamente. El capítulo 23 (sobre las bases de datos basadas en la lógica o deductivas) es una versión revisada del apéndice C anterior.
- *Parte VI:* El capítulo 24 es una versión totalmente revisada y muy mejorada de los capítu los 22 al 24 anteriores. El capítulo 25 es nuevo en su mayoría.

Por último, el apéndice A está basado en parte del antiguo capítulo 8, el apéndice B es nuevo y el apéndice C es una versión actualizada del apéndice D anterior.

Además de los cambios señalados arriba, en esta edición se incluyeron los temas siguientes:

- Estructuras de almacenamiento y métodos de acceso (apéndice A anterior);
- Exposición detallada de DB2 (apéndice B anterior).

¿QUÉ HACE A ESTE LIBRO DIFERENTE?

Todo libro de base de datos en el mercado tiene sus propias fortalezas y debilidades, y todo es-critor tiene sus propios intereses personales. Uno se concentra en los aspectos de administración; otro subraya el modelado entidad/vínculo; un tercero ve todo a través de un lente de SQL; otro más toma un punto de vista de "objeto" puro; otro ve el campo exclusivamente desde el punto de vista de los productos comerciales; y así sucesivamente. Y por supuesto, yo no soy la excep-ción a esa regla; también tengo un interés personal: que podría llamar el interés por los **funda-mentos.** Yo creo firmemente que debemos tener las bases adecuadas y comprenderlas bien, antes de intentar cualquier tipo de construcción sobre ellos. Esta creencia de mi parte explica el gran énfasis que pone este libro en el modelo relacional. En particular, esto explica lo extenso de la parte II —la parte más importante del libro— donde presento mi propia percepción del modelo relacional, tan detenidamente como puedo. Estoy interesado en los fundamentos, no en las novedades y modas.

En este ánimo, debo decir que me doy cuenta perfectamente de que el tono general de este libro ha cambiado a través de los años. Las primeras ediciones fueron en su mayoría de natu-raleza descriptiva: describían el campo como en realidad era en la práctica, "con todo y defec-tos". En contraste, esta edición es mucho más prescriptiva; habla de la forma en que debería ser el campo y en la que deberá desarrollarse en el futuro, si hacemos bien las cosas (en otras pa-labras, éste es un libro de texto ¡con un sentido! Y lo primero de "hacer bien las cosas" significa

en realidad educarse a uno mismo en lo que son las cosas correctas. Espero que esta edición pueda ayudar en ese esfuerzo.

Y otro punto: algunos de ustedes tal vez sepan que, junto con mi colega Hugh Darwen, publiqué recientemente otro libro "prescriptivo" sobre la tecnología de base de datos, cuyo título (abreviado) es *The Third Manifesto* [3.3]. Ese libro se basa en el modelo relacional para ofrecer una propuesta técnica detallada para los futuros sistemas de base de datos (es el resultado de muchos años de enseñanza y reflexión acerca de dichos aspectos, tanto por parte de Hugh como de la mía propia). Y sin que sea una sorpresa, las ideas del *Manifesto* anuncian el presente libro a todo lo largo. Lo cual no quiere decir que el *Manifesto* sea un prerrequisito para este libro; no le es, aunque sí es relevante de manera directa para gran parte de éste, y en él puede encontrar información adicional pertinente.

UN ULTIMO COMENTARIO

Quisiera cerrar estas notas introductorias con el siguiente extracto de otro prefacio; el del mismo Bertrand Russell a *The Bertrand Russell Dictionary of Mind, Mater and Morals* (ed., Lester E. Dennon), Citadel Press, 1993, reproducido aquí con autorización:

> *Se me ha acusado de tener el hábito de cambiar de opinión... De ninguna manera me avergüenzo [de ese hábito]. ¿Qué físico que estuviera activo en 1900 soñaría con jactarse de que sus opiniones no han cambiado durante el último medio siglo?... [La] clase de filosofía que valoro y me he esforzado por perseguir es científica, en el sentido de que existe cierto conocimiento definido por alcanzar y que esos nuevos descubrimientos pueden hacer inevitable la admisión de un error previo a cualquier mente candida. Por lo que he dicho, antes o después, yo no proclamo el tipo de verdad con la que los teólogos proclaman sus credos. Sólo afirmo, en el mejor de los casos, que la opinión expresada era una opinión sensible a sostener en ese momento... Me sorprendería mucho si la investigación posterior no mostrara que era necesario que fuera modificada. [Dichas opiniones no fueron] emitidas como pronunciamientos pontificios, sino como lo mejor que pude hacer en el momento por la promoción de un pensamiento claro y preciso. La claridad ha sido mi objetivo por encima de todo.*

Si compara las primeras ediciones de mi libro con esta séptima edición, encontrará que yo también he cambiado de opinión en muchos aspectos (y sin duda seguiré haciéndolo). Espero que usted acepte los comentarios arriba citados como una justificación adecuada para este estado de las cosas. Comparto la percepción de Bertrand Russell en relación con el campo de la investigación científica, pero él expresa esa percepción con mucha más elocuencia de lo que yo podría hacerlo.

RECONOCIMIENTOS

Una vez más, es un placer reconocer mi deuda con las muchas personas que participaron, en forma directa o indirecta, en la producción de este libro. En primer lugar, debo agradecer a mis amigos David McGoverán y Hugh Darwen por su importante colaboración en esta edición: David aportó el primer borrador del capítulo 21 sobre el apoyo para la toma de decisiones, y Hugh contribuyó con el primer borrador del capítulo 22 sobre bases de datos temporales. Hugh

realizó también un minucioso trabajo de revisión en gran parte del manuscrito, incluyendo en particular todos los capítulos del tema relacional y el apéndice sobre SQL3. En segundo lugar. el texto se benefició con los comentarios de muchos estudiantes de los seminarios que he venid impartiendo durante los últimos años. También se benefició en gran medida de los comentario y discusiones con diversos amigos y revisores, incluyendo a Charley Bontempo, Declan Brady. Hugh Darwen (de nuevo), Tim Hartley, Adrian Larner, Chung Lee, David Livingstone, Nikos Lorentzos, Hizha Lu, Ramon Mata-Toledo, Nelson Mattos, David McGoveran (otra vez), Fabian Pascal, Suda Ram, Rick van der Lans, Yongdong Wang, Colin White y Qiang Zhu. Cada una de estas personas revisaron por lo menos una parte del manuscrito, pusieron a mi disposición material técnico o me ayudaron a encontrar las respuestas a mis múltiples preguntas técnicas, por lo que les estoy muy agradecido. Por último, deseo (como siempre) dar las gracias a todos en Addison-Wesley —en especial a Maite Suarez-Rivas y a Katherine Harutunian— por su incentivo y apoyo a lo largo de este proyecto, y a mi editora, Elydia Davis, por su excelente trabajo de siempre.

Healdsburg, California
1999

C. J. Date

PARTE I

PRELIMINARES

La parte I consta de cuatro capítulos introductorios:

- El capítulo 1 prepara el escenario explicando qué es una base de datos y por qué son nece sarios los sistemas de bases de datos. También explica brevemente la diferencia entre los sistemas de bases de datos relaciónales y otros.
- A continuación, el capítulo 2 presenta una arquitectura general para sistemas de bases de datos, denominada *arquitectura ANSI/SPARC*. Dicha arquitectura sirve como una estructu ra sobre la cual se basará el resto del libro.
- Después, el capítulo 3 presenta un panorama general de los sistemas relaciónales (el obje tivo es que sirva como introducción para las explicaciones mucho más amplias de este tema que aparecen en la parte II y posteriores). También presenta y explica el ejemplo que va a ser utilizado en todo el libro, la base de datos de proveedores y partes.
- Por último, el capítulo 4 presenta el lenguaje relacional estándar SQL.

CAPITULO 1

Panorama general de la administración de bases de datos

1.1 INTRODUCCIÓN

Un sistema de **bases de datos** es básicamente un *sistema computarizado para llevar registros.* Es posible considerar a la propia **base de datos** como una especie de armario electrónico para archivar; es decir, es un depósito o contenedor de una colección de archivos de datos computarizados. Los usuarios del sistema pueden realizar una variedad de operaciones sobre dichos archivos. por ejemplo:

- Agregar nuevos archivos vacíos a la base de datos;
- Insertar datos dentro de los archivos existentes;
- Recuperar datos de los archivos existentes;
- Modificar datos en archivos existentes;
- Eliminar datos de los archivos existentes;
- Eliminar archivos existentes de la base de datos.

La figura 1.1 muestra una base de datos reducida que contiene un solo archivo, denominado CAVA, el cual contiene a su vez datos concernientes al contenido de una cava de vinos. La

NICHO#	VINO	PRODUCTOR	AÑO	BOTELLAS	LISTO
2	Chardonnay	Buena Vista	1997	1	1999
3	Chardonnay	Geyser Peak	1997	5	1999
6	Chardonnay	Simi	1996	4	1996
12	Joh. Riesling	Jekel	1998	1	1999
21	Fumé Blanc	Ch. St. Jean	1997	4	1999
22	Fumé Blanc	Robt. Mondavi	1996	2	1998
30	Gewurztraminer	Ch. St. Jean	1998	3	1999
43	Cab. Sauvignon	Windsor	1991	12	2000
45	Cab. Sauvignon	Geyser Peak	1994	12	2002
48	Cab. Sauvignon	Robt. Mondavi	1993	12	2004
50	Pinot Noir	Gary Farrel	1996	3	1999
51	Pinot Noir	Fetzer	1993	3	2000
52	Pinot Noir	Dehlinger	1995	2	1998
58	Merlot	Clos du Bois	1994	9	2000
64	Zinfandel	Cline	1994	9	2003
72	Zinfandel	Rafanelli	1995	2	2003

Figura 1.1 La base de datos de la cava de vinos (archivo CAVA).

Recuperación:

```
SELECT VINO, NICHO#, PRODUCTOR
FROM CAVA
WHERE LISTO = 2000
```

Resultado (por ejemplo, como se muestra en una pantalla de monitor):

VINO	NICHO#	PRODUCTOR
Cab. Sauvignon	43	Windsor Fetzer
Pinot Noir	51	Clos du Bois
Merlot	58	

Figura 1.2 Ejemplo de recuperación.

figura 1.2 muestra una operación de **recuperación** desde la base de datos, junto con los datos devueltos por dicha operación. *Nota:* Para una mayor claridad, a lo largo del libro mostramos en mayúsculas las operaciones de base de datos, los nombres de archivo y otro material similar. En la práctica es a menudo más conveniente escribir este material en minúsculas. La mayoría de los sistemas aceptan ambas denominaciones.

La figura 1.3 muestra ejemplos de operaciones de **inserción, modiñcación y eliminación** de la base de datos anterior que prácticamente se explican por sí mismos. Más adelante, en los capítulos 3,4,5, y en algunas otras partes, proporcionaré ejemplos de la incorporación y eliminación de archivos completos.

Inserción de datos nuevos:

```
INSERT
INTO CAVA ( NICHO*, VINO, PRODUCTOR, AÑO, BOTELLAS, LISTO )
VALUES ( 53, 'Pinot Noir', 'Saintsbury', 1997, 6, 2001 ) ;
```

Modificación de datos existentes:

```
UPDATE CAVA
SET   BOTELLAS • 4
WHERE NICHO# = 3 ;
```

Eliminación de datos existentes:

```
DELETE
FROM CAVA
WHERE NICHO# = 2 ;
```

Figura 1.3 Ejemplos de inserción, modificación y eliminación.

Puntos importantes de estos ejemplos:

1. Por razones obvias, a los archivos computarizados como el de CAVA de la figura 1.1.a menudo se les llama **tablas** (con más precisión, tablas **relaciónales.** Vea las secciones 1.3 y 1-6).
2. Podemos pensar en las **filas** de dicha tabla como los registros del archivo y en las **colum nas** como los campos de dichos registros. En este libro, emplearemos la terminología de registros y campos cuando hablemos de sistemas de base de datos en general (principal mente en los dos primeros capítulos); usaremos la terminología de filas y columnas cuando hablemos de sistemas relaciónales específicos (nuevamente, vea las secciones 1.3 y 1.6). *Nota:* En realidad, cuando abordemos explicaciones más formales en las partes posteriores del libro, cambiaremos a términos más formales.
3. Por razones de simplicidad, en el ejemplo anterior hicimos la suposición tácita de que las columnas VINO y PRODUCTOR contienen datos de tipo cadena de caracteres y que las demás columnas contienen datos enteros. Consideraremos con más detalle la cuestión de los **tipos de datos** de las columnas en los capítulos 3, 4 y en particular en el 5.
4. La columna NICHO# constituye la **clave primaria** de la tabla CAVA (lo que significa que no es posible que dos filas de CAVA contengan el mismo valor de NICHO#). A me nudo usamos un subrayado doble para señalar las columnas de clave primaria, como en la figura 1.1.
5. Las operaciones de ejemplo o "instrucciones" de las figuras 1.2 y 1.3 —SELECT, **INSERT, UPDATE, DELETE**— están expresadas en un lenguaje denominado.SQL. SQL es el len guaje estándar para interactuar con bases de datos relaciónales y es soportado por práctica mente todos los productos de base de datos actuales. *Nota:* El nombre "SQL" significaba originalmente "Lenguaje estructurado de consultas" y se pronunciaba "sikuel". Sin embar go, ahora que el lenguaje se ha convertido en un estándar, el nombre es solamente repre sentativo —no es oficialmente la abreviatura de nada— y la balanza se inclinó en favor de la pronunciación "es-kiu-el". En el libro tomaremos esta última pronunciación.
6. Observe que SQL utiliza la palabra clave UPDATE para indicar específicamente un "cam bio". Este hecho puede causar confusión, debido a que este término también solía referirse a las tres operaciones INSERT, UPDATE y DELETE como grupo. En este libro, usaremos la palabra "actualizar", en minúsculas, cuando nos refiramos al significado genérico y el operador UPDATE, en mayúsculas, cuando se trate de la operación específica de modi ficación.
7. Como es probable que ya sepa, la gran mayoría de sistemas de bases de datos actuales son relaciónales (o de todos modos deberían serlo —vea el capítulo 4, sección 4.7). En parte por esta razón, este libro hace énfasis en dichos sistemas.

Una última observación preliminar: la comprensión del material de este capítulo y el siguiente es fundamental para una apreciación completa de las características y capacidades de un sistema moderno de base de datos. Sin embargo, no puede negarse que el material es en cierto modo abstracto y un poco árido en ciertas partes, y que tiende a abarcar un gran número de conceptos y términos que podrían ser nuevos para usted. En las partes posteriores del libro —en especial en los capítulos 3 y 4— encontrará material mucho menos abstracto y por lo tanto quizás más comprensible. De ahí que tal vez prefiera, por el momento, dar a estos dos primeros

capítulos una "leída ligera" y volverlos a leer con más detenimiento cuando sean más relevantes para los temas que esté abordando.

1.2 ¿QUÉ ES UN SISTEMA DE BASE DE DATOS?

Para repetir lo que mencionamos en la sección anterior, un sistema de base de datos es básicamente un sistema computarizado para guardar registros; es decir, es un sistema computarizado cuya finalidad general es almacenar información y permitir a los usuarios recuperar y actualizar esa información con base en peticiones. La información en cuestión puede ser cualquier cosa que sea de importancia para el individuo u organización; en otras palabras, todo lo que sea necesario para auxiliarle en el proceso general de su administración.

Nota: en este libro los términos "datos" e "información" los trato como sinónimos. Algunos autores prefieren distinguir entre ambos, utilizando "datos" para referirse a lo que está en realidad almacenado en la base de datos e "información" para referirse al *significado* de esos datos como lo entiende algún usuario. La diferencia es importante; tan importante que parece preferible hacerla explícita donde sea necesario, en vez de depender de una diferenciación un tanto arbitraria entre dos términos que son en esencia sinónimos.

La figura 1.4 es una imagen simplificada de un sistema de base de datos. Pretende mostrar que un sistema de base de datos comprende cuatro componentes principales: **datos, hardware, software y usuarios.** A continuación consideramos brevemente estos cuatro componentes. Por

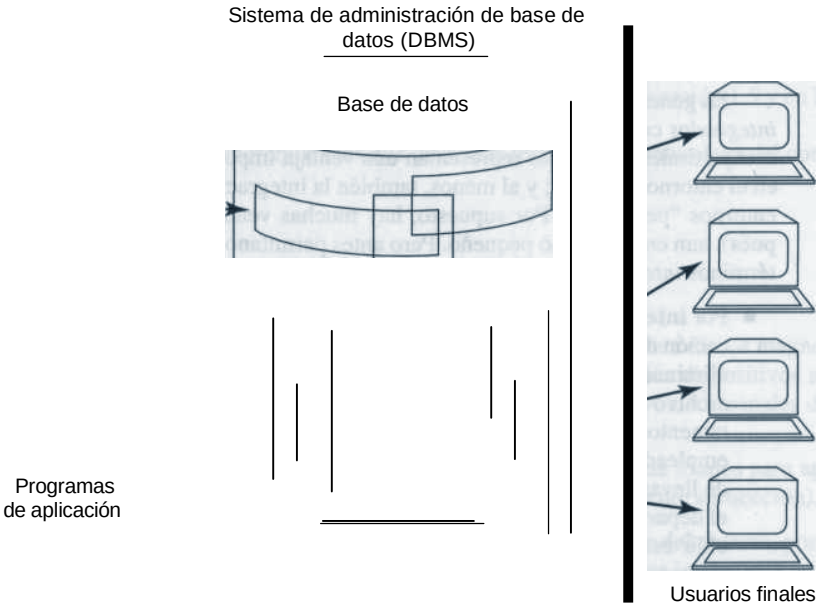


Figura 1.4 Imagen simplificada de un sistema de base de datos.

supuesto, más delante explicaremos cada uno con más detalle (con excepción del componente de hardware, cuyos detalles exceden en su mayoría el alcance de este libro).

Datos

Los sistemas de bases de datos están disponibles en máquinas que van desde las computadoras personales más pequeñas hasta las mainframes más grandes. Sobra decir que las facilidades que proporciona un sistema están determinadas hasta cierto punto por el tamaño y potencia de la máquina subyacente. En particular, los sistemas que se encuentran en máquinas grandes (s i s temas grandes") tienden a ser *multiusuario,* mientras que los que se ejecutan en máquinas pe queñas ("sistemas pequeños") tienden a ser *de un solo usuario.* Un **sistema de un solo usuario** es aquel en el que sólo un usuario puede tener acceso a la base de datos en un momento dado; un **sistema multiusuario** es aquel en el cual múltiples usuarios pueden tener acceso simultáneo a la base de datos. Como sugiere la figura 1.4, en este libro generalmente tomaremos el último caso; aunque de hecho la distinción es irrelevante en lo que respecta a la mayoría de los usuarios: En general, el objetivo principal en los sistemas multiusuario es precisamente permitir que cada usuario se comporte como si estuviera trabajando en un sistema *de un solo usuario.* **Los** problemas especiales de los sistemas multiusuario son en su mayoría problemas internos del sistema y no aquellos que son visibles al usuario (vea la parte IV de este libro, en especial el capítulo 15).

Nota: Para efectos de simplicidad, es conveniente suponer que la totalidad de los datos del sistema está almacenada en una sola base de datos, y en este libro haremos constantemente esta suposición (ya que materialmente no afecta ninguna de nuestras explicaciones posteriores). Sir embargo, en la práctica podría haber buenas razones (incluso en un sistema pequeño) para que los datos sean separados en diferentes bases de datos. Abordaremos algunas de estas razones mas adelante, en el capítulo 2 y en otras secciones.

En general, los datos de la base de datos —por lo menos en un sistema grande— serán tanto *integrados* como *compartidos.* Como veremos en la sección 1.4, los aspectos de integración y compartimiento de datos representan una ventaja importante de los sistemas de bases de datos en el entorno "grande"; y al menos, también la integración de datos puede ser importante en los entornos "pequeños". Por supuesto, hay muchas ventajas adicionales (que abordaremos después), aun en el entorno pequeño. Pero antes permítanos explicar lo que queremos decir con 1 términos *integrado* y *compartido.*

- Por **integrada,** queremos decir que podemos imaginar a la base de datos como una unifi-
  cación de varios archivos que de otro modo serían distintos, con una redundancia entre ellos eliminada al menos parcialmente. Por ejemplo, una base de datos dada podría contener un archivo EMPLEADO que proporcionara los nombres de los empleados, domicilios, depar tamentos, sueldos, etc. y un archivo INSCRIPCIÓN que representara la inscripción de los empleados a los cursos de capacitación (consulte la figura 1.5). Suponga ahora que, a fu de llevar a cabo el proceso de administración de cursos de capacitación, es necesario saber el departamento de cada estudiante inscrito. Entonces, resulta claro que no es necesario in cluir esa información de manera redundante en el archivo INSCRIPCIÓN, debido a que siempre puede consultarse haciendo referencia al archivo EMPLEADO.
- Por **compartida,** queremos decir que las piezas individuales de datos en la base pueden s compartidas entre diferentes usuarios y que cada uno de ellos puede tener acceso a la misma pieza de datos, probablemente con fines diferentes. Como indiqué anteriormente, distintos

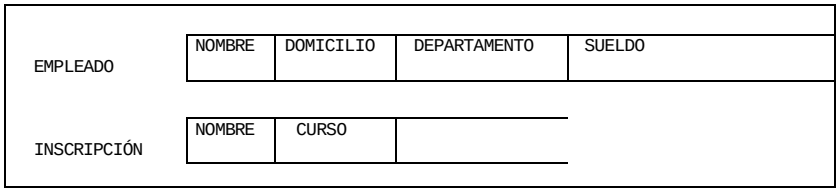


Figura 1.5 Los archivos EMPLEADO e INSCRIPCIÓN.

usuarios pueden en efecto acceder a la misma pieza de datos *al mismo tiempo* ("acceso concurrente"). Este compartimiento, concurrente o no, es en parte consecuencia del hecho de que la base de datos está integrada. En el ejemplo citado arriba, la información de departamento en el archivo EMPLEADO sería típicamente compartida por los usuarios del Departamento de personal y los usuarios del Departamento de capacitación; y como ya sugerí, estas dos clases de usuarios podrían emplear esa información para fines diferentes. *Nota:* en ocasiones, si la base de datos *no es* compartida, se le conoce como "personal" o como "específica de la aplicación".

Otra consecuencia de los hechos precedentes —que la base de datos sea integrada y (por lo regular) compartida— es que cualquier usuario ocupará normalmente sólo una pequeña parte de la base de datos total; lo que es más, las partes de los distintos usuarios se traslaparán de diversas formas. En otras palabras, una determinada base de datos será percibida de muchas formas diferentes por los distintos usuarios. De hecho, aun cuando dos usuarios tengan la misma porción de la base de datos, su visión de dicha parte podría diferir considerablemente a un nivel detallado. Este último punto lo explico en forma más completa en la sección 1.5 y en los capítulos 2, 3 y en especial el 9.

En la sección 1.3 tendremos más qué decir con respecto a la naturaleza del componente de datos del sistema.

Hardware

Los componentes de hardware del sistema constan de:

- Los volúmenes de almacenamiento secundario —principalmente discos magnéticos— que se emplean para contener los datos almacenados, junto con los dispositivos asociados de E/S (unidades de discos, etc.), los controladores de dispositivos, los canales de E/S, entre otros; y
- Los procesadores de hardware y la memoria principal asociada usados para apoyar la eje cución del software del sistema de base de datos (vea la siguiente subsección).

Este libro no hace mucha referencia a los aspectos de hardware del sistema, por las siguientes razones (entre otras): Primero, estos aspectos conforman un tema importante por sí mismos; segundo, los problemas que se encuentran en esta área no son exclusivos de los sistemas de base de datos; y tercero, dichos problemas han sido investigados en forma minuciosa y descritos en otras partes.

Software

Entre la base de datos física —es decir, los datos como están almacenados físicamente— usuarios del sistema, hay una capa de software conocida de manera indistinta como el administrador de base de datos o el servidor de base de datos; o más comúnmente como el sistema de administración de base de datos (DBMS). Todas las solicitudes de acceso a la base de datos son manejadas por el DBMS; las características que esbozamos en la sección 1.1 para agre eliminar archivos (o tablas), recuperar y almacenar datos desde y en dichos archivos, etcétera son características que proporciona el DBMS. Por lo tanto, una función general que ofrece DBMS consiste en *ocultar a los usuarios de la base de datos los detalles al nivel de hará* (en forma muy parecida a como los sistemas de lenguajes de programación ocultan a los programadores de aplicaciones los detalles a nivel de hardware). En otras palabras, el DBMS ofrece a los usuarios una percepción de la base de datos que está, en cierto modo, por encima del nivel del hardware y que maneja las operaciones del usuario (como las operaciones SQL explicad brevemente en la sección 1.1) expresadas en términos de ese nivel más alto de percepción. A lo largo de este libro explicaremos con mayor detalle ésta y otras funciones del DBMS. Algunos aspectos adicionales:

- El DBMS es, por mucho, el componente de software más importante del sistema en gene ral, aunque no es el único. Otros comprenden las utilerías, herramientas de desarrollo de aplicaciones, ayudas de diseño, generadores de informes y (el más importante) el *administrador de transacciones* o *monitor PT.* Para una mayor explicación de estos componentes. consulte los capítulos 2 y 3 y (en especial) la parte IV.
- El término *DBMS* se usa también para referirse en forma genérica a un producto determin de algún fabricante; por ejemplo, el producto "DB2 Universal Database" de IBM para OS/390. El término *ejemplar de DBMS* se usa entonces para referirse a una copia de dicho producto que opera en alguna instalación de computadora determinada. Como seguram notará, en ocasiones es necesario distinguir cuidadosamente entre estos dos conceptos.

  Nota: Debemos advertirle que en la industria de las computadoras la gente a menudo usa el término *base de datos* cuando en realidad se refieren al *DBMS* (en cualquiera de lo sentidos anteriores). He aquí un ejemplo típico: "El fabricante de la base de datos X superó al fabricante de la base de datos *Y* en proporción de dos a uno." Este uso es engañoso y no es correcto, aunque es mucho muy común. (Por supuesto, el problema es que si al DBMS lo llamamos base de datos, entonces ¿cómo llamaremos a la base de datos?) *Advertencia para el lector.*

Usuarios

Consideramos tres grandes clases de usuarios (y que en cierto modo se traslapan):

- Primero, hay programadores de aplicaciones responsables de escribir los programas de aplicación de base de datos en algún lenguaje de programación como COBOL. PL/1. C++ Java o algún lenguaje de alto nivel de la "cuarta generación" (vea el capítulo 2). Estos pro-gramas acceden a la base de datos emitiendo la solicitud apropiada al DBMS (por lo regu lar una instrucción SQL). Los programas en sí pueden ser aplicaciones convenciona por lotes o pueden ser aplicaciones en línea, cuyo propósito es permitir al usuario final el acceso a la base de datos desde una estación de trabajo o terminal en línea (vea el parrafo siguiente). Las aplicaciones más modernas pertenecen a esta variedad.

- En consecuencia, la segunda clase de usuarios son los **usuarios finales,** quienes interactúan con el sistema desde estaciones de trabajo o terminales en línea. Un usuario final puede acceder a la base de datos a través de las aplicaciones en línea mencionadas en el párrafo anterior, o bien puede usar una interfaz proporcionada como parte integral del software del sistema de base de datos. Por supuesto, las interfaces proporcionadas por el fabricante están apoyadas también por aplicaciones en línea, aunque esas aplicaciones están **integradas;** es decir, no son escritas por el usuario. La mayoría de los sistemas de base de datos incluyen por lo menos una de estas aplicaciones integradas, digamos un **procesador de lenguaje de consulta,** mediante el cual el usuario puede emitir solicitudes a la base de datos (también conocidas como instrucciones o *comandos),* como SELECT e INSERT, en forma interactiva con el DBMS. El lenguaje SQL mencionado en la sección 1.1 es un ejemplo típico de un lenguaje de consulta de base de datos.

  Nota: El término "lenguaje de consulta", a pesar de ser común, no es muy preciso, ya que el verbo "consultar" en lenguaje normal sugiere sólo una *recuperación,* mientras que los lenguajes de consulta por lo regular (aunque no siempre) ofrecen también actualización y otras operaciones.

  La mayoría de los sistemas proporcionan además interfaces integradas adicionales en las que los usuarios no emiten en absoluto solicitudes explícitas a la base de datos, como SELECT, sino que en vez de ello operan mediante (por ejemplo) la selección de elementos en un menú o llenando casillas de un formulario. Estas interfaces **controladas por menus** o **por formularios** tienden a facilitar el uso a personas que no cuentan con una capacitación formal en IT (Tecnología de la información; la abreviatura IS, de Sistemas de información, también es muy usada con el mismo significado). En contraste, las **interfaces controladas por comandos** (por ejemplo, los lenguajes de consulta) tienden a requerir cierta experiencia profesional en IT, aunque tal vez no demasiada (obviamente no tanta como la que es necesaria para escribir un programa de aplicación en un lenguaje como COBOL). Por otra parte, es probable que una interfaz controlada por comandos sea más flexible que una controlada por menus o por formularios, dado que los lenguajes de consulta por lo regular incluyen ciertas características que no manejan esas otras interfaces.
- El tercer tipo de usuario, que no aparece en la figura 1.4, es el **administrador de base de datos** o DBA. La función del DBA, y la función asociada (muy importante) del administrador de **datos,** se abordará en la sección 1.4 y en el capítulo 2 (sección 2.7).

Con esto concluimos nuestra descripción preliminar de los aspectos más importantes de un sistema de base de datos. Ahora continuaremos con la explicación de estas ideas con un poco más de detalle.

1.3 ¿QUE ES UNA BASE DE DATOS?

Datos persistentes

Es una costumbre referirse a los datos de la base de datos como "persistentes" (¡aunque en realidad éstos podrían no persistir por mucho tiempo!). Por *persistentes* queremos decir, de manera intuitiva, que el tipo de datos de la base de datos difiere de otros datos más efímeros, como los datos de entrada, los datos de salida, las instrucciones de control, las colas de trabajo, los bloques de control de software, los resultados intermedios y de manera más general, cualquier dato que sea de naturaleza transitoria. En forma más precisa, decimos que los datos de la base de datos "persisten" debido

en primer lugar a que una vez aceptados por el DBMS para entrar en la base de datos, *en lo sucesivo sólo pueden ser removidos de la base de datos por alguna solicitud explícita al DBMS,* no como un mero efecto lateral de (por ejemplo) algún programa que termina su ejecución. Por lo tanto, esta noción de persistencia nos permite dar una definición más precisa del término "base de datos":

■ Una **base de datos** es un conjunto de datos persistentes que es utilizado por los sistemas de aplicación de alguna empresa dada.

Aquí, el término "empresa" es simplemente un término genérico conveniente para identificar a cualquier organización independiente de tipo comercial, técnico, científico u otro. Una empresa podría ser un solo individuo (con una pequeña base de datos personal), toda una corporación o un gran consorcio similar (con una gran base de datos compartida) o todo lo que se ubique entre estas dos opciones. Aquí tenemos algunos ejemplos:

1. Una compañía manufacturera
2. Un banco
3. Un hospital
4. Una universidad
5. Un departamento gubernamental

Toda empresa necesariamente debe mantener una gran cantidad de datos acerca de su operación. Estos datos son los "datos persistentes" a los que nos referimos antes. En forma característica, las empresas que acabamos de mencionar incluirían entre sus datos persistentes a los siguientes:

1. Datos de producción
2. Datos contables
3. Datos de pacientes
4. Datos de estudiantes
5. Datos de planeación

Nota: Las primeras ediciones de este libro utilizaban el término "datos operacionales" en lugar de "datos persistentes". El primer término reflejaba el énfasis original en los sistemas de base de datos con aplicaciones **operacionales** o **de producción;** es decir, aplicaciones rutinarias altamente repetitivas que eran ejecutadas una y otra vez para apoyar la operación cotidiana de la empresa (por ejemplo, una aplicación para manejar los depósitos o retiros de efectivo en un sistema bancario). Para describir este tipo de entorno, se ha llegado a utilizar el término **procesamiento de transacciones en línea.** Sin embargo, ahora las bases de datos se utilizan cada vez más también para otro tipo de aplicaciones (por ejemplo, aplicaciones de **apoyo a la toma de decisiones)** y el término "datos operacionales" ya no es del todo apropiado. De hecho, hoy en días las empresas mantienen generalmente dos bases de datos independientes; una que contiene los datos operacionales y otra, a la que con frecuencia se le llama *almacén de datos (data warehouse),* que contiene datos de apoyo para la toma de decisiones. A menudo el almacén de datos incluye *información de resumen* (por ejemplo, totales, promedios...) y dicha información a su vez se extrae periódicamente de la base de datos operacional; digamos una vez al día o una vez por semana. Para una explicación más amplia de las bases de datos y las aplicaciones de apoyo a la toma de decisiones, consulte el capítulo 21.

Entidades y vínculos

Ahora consideraremos el ejemplo de una compañía manufacturera ("KnowWare Inc.") con un poco más de detalle. Por lo general, una empresa así desea registrar la información sobre los *proyectos* que maneja, las *partes* que utiliza en dichos proyectos, los *proveedores* que suministran esas partes, los *almacenes* en donde guardan esas partes, los *empleados* que trabajan en esos proyectos, etcétera. Por lo tanto los proyectos, partes, proveedores, etcétera, constituyen las **entidades** básicas de información que KnowWare Inc. necesita registrar (el término "entidad" es empleado comúnmente en los círculos de bases de datos para referirse a cualquier objeto distinguible que va a ser representado en la base de datos). Vea la figura 1.6.

Además de las propias entidades básicas (como los proveedores, las partes, etcétera, en el ejemplo), habrá también **vínculos** que asocian dichas entidades básicas. Estos vínculos están representados por los rombos y las líneas de conexión de la figura 1.6. Por ejemplo, existe un vínculo ("VP") entre proveedores y partes: cada proveedor suministra ciertas partes y de manera inversa, cada parte es suministrada por ciertos proveedores (para ser más precisos, cada proveedor suministra ciertos *tipos* de partes y cada *tipo* de parte es suministrado por ciertos proveedores). En forma similar, las partes son utilizadas en proyectos y de manera inversa, los proyectos utilizan partes (vínculo PY); las partes son guardadas en almacenes y los almacenes guardan partes (vínculo AP); y así sucesivamente. Observe que todos estos vínculos son *bidireccionales;* es decir, pueden ser recorridos en ambas direcciones. Por ejemplo, el vínculo VP entre proveedores y partes puede ser usado para responder las dos siguientes preguntas:

- Dado un proveedor, obtener las partes que éste suministra.
- Dada una parte, obtener los proveedores que la suministran.

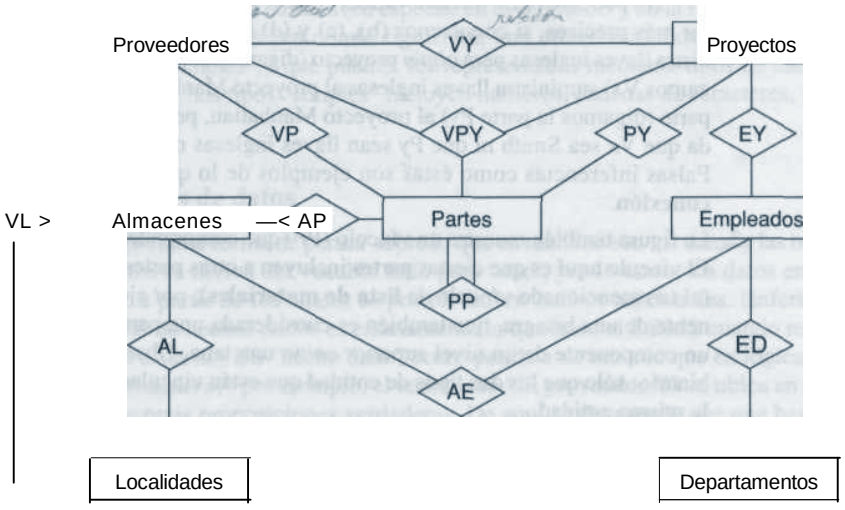


Figura 1.6 Diagrama de entidad/vínculo (E/R) para KnowWare Inc.

El punto importante con respecto a este vínculo (y por supuesto, con respecto a todos los vínculos de la figura) es que *son parte de los datos tanto como lo son las entidades básicas.* Por lo tanto, deben estar representados en la base de datos al igual que las entidades básicas. ***Nota:*** Como veremos en el capítulo 3, específicamente en un sistema relacional, tanto las **entidades** básicas como los vínculos que las conectan serán representados por medio de tablas como la que se muestra en la figura 1.1.

Observemos de paso que la figura 1.6 es un ejemplo de lo que se denomina (por razones obvias) **diagrama de entidad/vínculo** (diagrama E/R para abreviar). En el capítulo 13 consideraremos estos diagramas con un poco más de detalle.

La figura 1.6 ilustra además otros puntos importantes:

1. Aunque la mayoría de los vínculos de la figura comprenden *dos* tipos de entidad (es decir. son vínculos *binarios)* no significa que todos los vínculos deban ser necesariamente bina rios en este aspecto. En el ejemplo hay un vínculo ("VPY") que involucra tres tipos de en tidad (proveedores, partes y proyectos): un vínculo *ternario.* La interpretación que pretendo dar es que ciertos proveedores suministran ciertas partes para ciertos proyectos. Observe con cuidado que este vínculo ternario ("los proveedores suministran partes para proyectos") normalmente *no* equivale a la combinación de tres vínculos binarios "los proveedores sumi nistran partes", "las partes se usan en proyectos" y "los proyectos son abastecidos por los proveedores". Por ejemplo, la declaración de que

   (a) Smith suministra llaves inglesas para el proyecto Manhattan,

   nos dice *más* de lo que expresan las tres declaraciones siguientes:

   (b) Smith suministra llaves inglesas,

   (c) Las llaves inglesas se usan en el proyecto Manhattan y

   (d) El proyecto Manhattan es abastecido por Smith

   No podemos (¡de manera válida!) inferir (a) conociendo únicamente (b). (c) y (d). Para ser más precisos, si conocemos (b), (c) y (d) entonces podríamos inferir que Smith suministra llaves inglesas para *algún* proyecto (digamos el proyecto Yz), que *cierto* proveedor (digamos Vx) suministra llaves inglesas al proyecto Manhattan, y que Smith suministra *alguna* parte (digamos la parte Py) al proyecto Manhattan, pero no podemos inferir en forma válida que V*x* sea Smith ni que Py sean llaves inglesas ni que Yz sea el proyecto Manhattan. Falsas inferencias como éstas son ejemplos de lo que a veces se denomina **trampa de conexión.**

2. La figura también muestra un vínculo (PP) que comprende sólo *un* tipo de entidad (partes). El vínculo aquí es que ciertas partes incluyen a otras partes como componentes inmediatos (el tan mencionado vínculo de **lista de materiales);** por ejemplo, un tornillo es un compo nente de una bisagra, que también es considerada una parte y podría ser a su vez parte de un componente de un nivel superior como una tapa. Observe que el vínculo sigue siendo binario; sólo que los dos tipos de entidad que están vinculados (partes y partes) vienen a ser la misma entidad.

3. En general, un conjunto determinado de tipos de entidad podría vincularse entre sí en cualquier cantidad de vínculos distintos. En el ejemplo de la figura 1.6, hay dos vínculos distintos que involucran a proyectos y empleados: Uno (EY) representa el hecho de que los empleados están asignados a proyectos, el otro (MY) representa el hecho de que los em pleados administran proyectos.

Ahora observamos que *un vínculo puede considerarse como una entidad por derecho propio.* Si tomamos nuestra definición de entidad como "cualquier objeto acerca del cual queremos registrar información", entonces un vínculo se ajusta perfectamente a la definición. Por ejemplo, "la parte P4 está guardada en el almacén A8" es una entidad acerca de la cual bien querríamos registrar información (por ejemplo, la cantidad correspondiente). Más aún, podemos obtener ventajas definitivas (que exceden el alcance de este capítulo) no haciendo distinciones innecesarias entre entidades y vínculos. Por lo tanto, en este libro trataremos en su mayoría a los vínculos sólo como una clase especial de entidad.

Propiedades

Como acabamos de señalar, una entidad es cualquier objeto acerca del cual queremos registrar información. De donde se desprende que las entidades (incluidos los vínculos) poseen **propiedades** que corresponden a la información que deseamos registrar sobre ellas. Por ejemplo, los proveedores tienen *localidades;* las partes tienen *pesos;* los proyectos tienen *prioridades;* las asignaciones (de empleados a proyectos) tienen *fechas de inicio,* etcétera. Por lo tanto, dichas propiedades deben estar representadas en la base de datos. Por ejemplo, la base de datos podría incluir una tabla denominada V que represente a los proveedores y esa tabla podría incluir una columna de nombre CIUDAD que represente a las localidades de los proveedores.

En general, las propiedades pueden ser tan simples o tan complejas como queramos. Por ejemplo, la propiedad "localidad del proveedor" es supuestamente bastante simple, ya que sólo consiste en un nombre de ciudad y puede ser representada en la base de datos por una simple cadena de caracteres. En contraste, un almacén podría tener una propiedad "plan de piso", que podría ser bastante compleja, consistir tal vez en todo un dibujo arquitectónico y en el texto descriptivo asociado. Al momento de la publicación de este libro, la mayoría de los productos de bases de datos estaban apenas logrando manejar propiedades complejas como el dibujo y el texto. Regresaremos a este tema más delante (en especial en el capítulo 5 y en la Parte VI); mientras tanto, en la mayoría de los casos (en donde signifique una diferencia) daremos por hecho que las propiedades son "simples" y que pueden ser representadas mediante tipos de datos "simples". Los ejemplos de dichos tipos "simples" incluyen números, cadenas de caracteres, fechas, horas, etcétera.

Datos y modelos de datos

Existe otra importante forma de pensar sobre lo que en realidad son los datos y las bases de datos. La palabra *datos* se deriva del vocablo latín para "dar"; por lo tanto, los datos en realidad son *hechos dados,* a partir de los cuales es posible inferir hechos adicionales. (Inferir hechos adicionales a partir de hechos dados es exactamente lo que hace el DBMS cuando responde a una consulta de un usuario.) Un "hecho dado" corresponde a su vez a lo que en lógica se denomina *proposición verdadera;** por ejemplo, el enunciado "El proveedor V1 se ubica en Londres" podría ser una de estas proposiciones verdaderas. De aquí se desprende que una base de datos es en realidad *una colección de tales proposiciones verdaderas* [1.2].

---

*En lógica, una proposición es algo que se evalúa ya sea como *verdadero* o como *falso* en forma inequívoca. Por ejemplo, "William Shakespeare escribió *Pride and Prejudice"* es una proposición (por cierto, una falsa).

Una razón por la que los sistemas de bases de datos relaciónales se han vuelto tan dominantes, tanto en el mundo industrial como en el académico, es que manejan en forma muy directa (de hecho casi trivial) la interpretación precedente de los datos y las bases de datos. Los sistemas relaciónales están basados en una teoría formal denominada **el modelo de datos relacional,** de acuerdo con el la cual:

- En tablas, los datos son representados por medio de filas y estas filas pueden interpretarse directamente como proposiciones verdaderas. Por ejemplo, en la figura 1.1, la fila NICHO# 72 puede interpretarse en forma obvia como la siguiente proposición verdadera:

  "El nicho número 72 contiene dos botellas de Zinfandel Rafanelli 1995, y estarán listas para su consumo en el año 2003."

- Se proporcionan operadores para operar sobre las columnas de las tablas, y estos operadores soportan directamente el proceso de inferir proposiciones verdaderas adicionales a partir de las ya dadas. Como ejemplo sencillo, el operador relacional *proyectar* (vea la sección 1.6) nos permite inferir, a partir de la proposición verdadera que acabamos de citar, la siguiente proposición verdadera, entre otras:

  "Algunas botellas de Zinfandel estarán listas para su consumo en el año 2003."

  (para ser más precisos: "Algunas botellas de Zinfandel, en algún nicho, producidas por algún productor en algún año, estarán listas para su consumo en el año 2003.")

Sin embargo, el modelo relacional no es el único modelo de datos. Existen otros (vea la sección 1.6), aunque la mayoría de ellos difieren del modelo relacional en que son hasta cierto grado *específicos,* en vez de estar basados firmemente en la lógica formal. Sea lo que fuere, surge la pregunta: ¿En general qué *es* un modelo de datos? Podemos definir el concepto como sigue:

> Un **modelo de datos** es una definición lógica, independiente y abstracta de los objetos, operadores y demás que en conjunto constituyen la *máquina abstracta* con la que interactúan los usuarios. Los objetos nos permiten modelar la *estructura* de los datos. Los operadores nos permiten modelar su *comportamiento.*

Entonces, de manera útil, podemos distinguir al modelo de su *implementacion:*

- La **implementación** de determinado modelo de datos es una realización física, en una máqui na real, de los componentes de la máquina abstracta que en conjunto constituyen ese modelo.

Resumiendo: El modelo es aquello que los usuarios tienen que conocer; la implementación es lo que los usuarios no tienen que conocer.

Nota: Como puede ver, la distinción entre modelo e implementación es en realidad sólo un caso especial (uno muy importante) de la conocida distinción entre *lógico y físico.* Sin embargo, por desgracia muchos de los sistemas de bases de datos actuales (incluso aquellos que dicen ser relaciónales) no hacen estas distinciones con tanta claridad como debieran. De hecho, parece no haber un buen entendimiento de estas distinciones y de la importancia de hacerlas. Como consecuencia, a menudo hay una brecha entre los *principios* de las bases de datos (la forma en que los sistemas de bases de datos deberían funcionar) y la *práctica* de las bases de datos (la forma en que realmente funcionan). En este libro nos enfocamos principalmente en los principios; aunque es justo advertirle que cuando comience a utilizar un producto comercial, podría llevarse algunas sorpresas desagradables.

Para concluir esta sección, debemos mencionar el hecho de que en realidad el término *modelo de datos* es utilizado en la literatura con dos significados muy distintos. El primero es como se describió anteriormente. El segundo es como un modelo de los datos persistentes de alguna *empresa en particular* (por ejemplo, la compañía manufacturera KnowWare Inc. que mencionamos anteriormente en esta sección). La diferencia entre ambos significados puede ser caracterizada como sigue:

- En el primer sentido, un modelo de datos es como un *lenguaje de programación* (aunque en cierto modo abstracto) cuyos elementos pueden ser usados para resolver una amplia va riedad de problemas específicos, pero que en sí y por sí mismos no tienen una conexión di recta con ninguno de estos problemas específicos.
- En el segundo sentido, un modelo de datos es como un *programa específico* escrito en ese lenguaje. En otras palabras, un modelo de datos que toma las características que ofrece al gún modelo como el primero y las aplica a cierto problema específico. Puede ser visto como una *aplicación específica* de algún modelo con el primer significado.

De aquí en adelante, en este libro usaremos el término *modelo de datos* sólo en el primer sentido, excepto cuando se indique lo contrario.

1.4 ¿POR QUÉ UNA BASE DE DATOS?

¿Por qué utilizar un sistema de base de datos? ¿Cuáles son las ventajas? Hasta cierto punto, la respuesta a estas preguntas depende de si el sistema en cuestión es de un solo usuario o multiusuario (o para ser más precisos, existen muchas ventajas *adicionales* en el caso del sistema multiusuario). Consideraremos primero el caso de un solo usuario.

Vuelva a ver el ejemplo de la cava de vinos de la figura 1.1, el cual puede ser ilustrativo para el caso de un solo usuario. Ahora bien, esa base de datos es tan reducida y sencilla que las ventajas podrían no ser tan obvias. Pero imagine una base de datos similar para un gran restaurante, con una existencia tal vez de miles de botellas y cambios muy frecuentes a dichas existencias; o piense en una tienda de licores, también con una gran existencia y una mayor rotación en la misma. Tal vez en estos casos sea más fácil apreciar las ventajas de un sistema de base de datos sobre los métodos tradicionales basados en papel, para llevar un registro. He aquí algunas:

- *Compactación:* No hay necesidad de archivos en papel voluminosos.
- *Velocidad:* La máquina puede recuperar y actualizar datos más rápidamente que un humano. En particular, las consultas *específicas* sin mucha elaboración (por ejemplo, "¿Tenemos más Zinfandel que Pinot Noir?") pueden ser respondidas con rapidez, sin necesidad de búsquedas manuales o visuales que llevan tiempo.
- *Menos trabajo laborioso;* Se puede eliminar gran parte del trabajo de llevar los archivos a mano. Las tareas mecánicas siempre las realizan mejor las máquinas.
- *Actualidad:* En el momento que la necesitemos, tendremos a nuestra disposición información precisa y actualizada.

Desde luego, los beneficios anteriores se aplican aún con más fuerza en un entorno multiusuario, donde es probable que la base de datos sea mucho más grande y compleja que en el caso

de un solo usuario. No obstante, en el entorno multiusuario hay una ventaja adicional, que expresaremos así: *El sistema de base de datos ofrece a la empresa un control centralizado de sus datos* (los cuales, como se habrá dado cuenta a estas alturas, constituyen uno de sus activos más valiosos). Esta situación contrasta en gran medida con la que se encuentra en una empresa que no cuenta con un sistema de base de datos, en donde por lo general cada aplicación tiene sus propios archivos privados (a menudo también sus propias cintas y discos) de modo que los datos están muy dispersos y son difíciles de controlar de una forma sistemática.

Administración de datos y administración de bases de datos

Explicaremos brevemente este concepto del control centralizado. El concepto implica que en la empresa habrá alguna persona identificable que tendrá esta responsabilidad central sobre los datos. Esa persona es el **administrador de datos** (o DA) que mencionamos brevemente al final de la sección 1.2. Ya que (repitiendo) los datos son uno de los activos más valiosos de la empresa, es imperativo que exista una persona que los entienda junto con las necesidades de la empresa con respecto a esos datos, *a un nivel de administración superior.* Esa persona es el administrador de datos. Por lo tanto, es labor del administrador de datos decidir en primer lugar qué datos deben ser almacenados en la base de datos y establecer políticas para mantener y manejar esos datos una vez almacenados. Un ejemplo de estas políticas podría ser una que indicara quién puede realizar qué operaciones sobre ciertos datos y bajo qué circunstancias. En otras palabras, una política de *seguridad de los datos* (vea la siguiente subsección).

Hay que destacar que el administrador de datos es un administrador, no un técnico (aunque es cierto que necesita tener cierta idea de las posibilidades que tienen los sistemas de base de datos en el ámbito técnico). El *técnico* responsable de implementar las decisiones del administrador de datos es el **administrador de** *base* **de datos** (o DBA). Por lo tanto, el DBA, a diferencia del administrador de datos, es un *profesional IT.* El trabajo del DBA consiste en crear la base de datos real e implementar los controles técnicos necesarios para hacer cumplir las diversas decisiones de las políticas hechas por el administrador de datos. El DBA también es responsable de asegurar que el sistema opere con el rendimiento adecuado y de proporcionar una variedad de otros servicios técnicos. Por lo regular, el DBA tendrá un equipo de programadores de sistemas y otros asistentes técnicos (es decir, en la práctica la función del DBA normalmente es realizada por un equipo de personas, no por una sola); sin embargo, para fines de simplicidad, es conveniente suponer que el DBA es de hecho un solo individuo. En el capítulo 2 abordaremos con más detalle la función del DBA.

Beneficios del enfoque de base de datos

En esta subsección identificaremos algunas de las ventajas específicas que surgen de la noción anterior de control centralizado.

- *Los datos pueden compartirse*

  Explicamos este punto en la sección 1.2, pero para complementar lo mencionaremos de nuevo aquí. Compartir no sólo significa que las aplicaciones existentes puedan compartir la información de la base de datos, sino también que sea posible desarrollar nuevas aplicaciones para operar sobre los mismos datos. En otras palabras, es posible satisfacer los

requerimientos de datos de aplicaciones nuevas sin tener que agregar información a la base de datos.

Es posible reducir la redundancia

En sistemas que no son de bases de datos, cada aplicación tiene sus propios archivos exclusivos. A menudo este hecho puede conducir a una redundancia considerable de los datos almacenados, con el consecuente desperdicio de espacio de almacenamiento. Por ejemplo, una aplicación de personal y una aplicación de registro de escolaridad podrían tener un archivo que tuviera información departamental de los empleados. Sin embargo, como sugerí en la sección 1.2, estos dos archivos pueden integrarse y eliminar así la redundancia, en tanto el administrador de datos esté consciente de los requerimientos de datos de ambas aplicaciones; es decir, en tanto la empresa tenga el control general necesario.

Por cierto, no pretendemos sugerir que *toda* la redundancia pueda o deba necesariamente ser eliminada. En ocasiones hay razones sólidas, tácticas o del negocio, para mantener varias copias distintas de los mismos datos. Sin embargo, sí pretendemos sugerir que cualquier redundancia de este tipo debe ser **controlada** cuidadosamente; es decir, el DBMS debe estar al tanto de ella, si es que existe, y debe asumir la responsabilidad de "propagar las actualizaciones" (vea el siguiente punto).

Es posible (hasta cierto grado) evitar la inconsistencia

Éste es en realidad el corolario del punto anterior. Suponga que un hecho más real —digamos que el empleado E3 trabaja en el departamento D8— está representado por dos entidades distintas en la base de datos. Suponga también que el DBMS no está al tanto de esta duplicidad (es decir, la redundancia no está controlada). Entonces necesariamente habrá ocasiones en las que las dos entidades no coincidan: digamos, cuando una de ellas ha sido actualizada y la otra no. En esos momentos, decimos que la base de datos es *inconsistente.* Resulta claro que una base de datos en un estado inconsistente es capaz de proporcionar a sus usuarios información incorrecta o contradictoria.

Por supuesto, si el hecho anterior es representado por una sola entrada (es decir, si se elimina la redundancia), entonces no puede ocurrir tal inconsistencia. Como alternativa; si no se elimina la redundancia pero se *controla* (haciéndola del conocimiento del DBMS), entonces el DBMS puede garantizar que la base de datos nunca será inconsistente *a los ojos del usuario,* asegurando que todo cambio realizado a cualquiera de las dos entidades será aplicado también a la otra en forma automática. A este proceso se le conoce como **propagación de actualizaciones.**

Es posible brindar un manejo de transacciones

Una **transacción** es una unidad de trabajo lógica, que por lo regular comprende varias operaciones de la base de datos (en particular, varias operaciones de actualización). El ejemplo común es el de transferir una cantidad de efectivo de una cuenta A a otra cuenta *B.* Es claro que aquí se necesitan dos actualizaciones, una para retirar el efectivo de la cuenta A y la otra para depositarlo en la cuenta *B.* Si el usuario declara que las dos actualizaciones son parte de la misma transacción, entonces el sistema puede en efecto garantizar que se hagan ya sea ambas o ninguna de ellas, aun cuando el sistema fallara (digamos por falta de suministro eléctrico) a la mitad del proceso.

Nota: La característica de *atomicidad* de las transacciones que acabamos de ilustrar no es el único beneficio del manejo de transacciones, pero a diferencia de las otras característi-

cas, ésta se aplica aun en el caso de un solo usuario.* En los capítulos 14 y 15 aparece una descripción completa de las diversas ventajas del manejo de transacciones y de cómo pueden ser logradas.

Es posible mantener la integridad

El problema de la integridad es el de asegurar que los datos de la base de datos estén correctos. La inconsistencia entre dos entradas que pretenden representar el mismo "hecho" es un ejemplo de la falta de integridad (vea antes la explicación de este punto, en esta subsección); desde luego, este problema en particular puede surgir sólo si existe redundancia en los datos almacenados. No obstante, aun cuando no exista redundancia, la base de datos podría seguir conteniendo información incorrecta. Por ejemplo, un empleado podría aparecer con 400 horas laboradas durante la semana, en lugar de 40; o como parte de un departamento que no existe. El control centralizado de la base de datos puede ayudar a evitar estos problemas (en la medida de lo posible) permitiendo que el administrador de datos defina y el DBA implemente las **restricciones de integridad** (también conocidas como *reglas del negocio)* que serán verificadas siempre que se realice una operación de actualización.

Vale la pena señalar que la integridad de los datos es aún más importante en un sistema de base de datos que en un entorno de "archivos privados", precisamente porque los datos son compartidos. Sin los controles apropiados sería posible que un usuario actualizara la base de datos en forma incorrecta, generando así datos malos e "infectando" a otros usuarios con esos datos. También debemos mencionar que actualmente la mayoría de los productos de bases de datos son mas bien débiles con respecto al manejo de las restricciones de integridad (aunque ha habido algunas mejoras recientes en esta área). Éste es un hecho desafortunado, ya que (como veremos en el capítulo 8) las restricciones de integridad son fundamentales y de crucial importancia, mucho más de lo que por lo regular apreciamos.

Es posible hacer cumplir la seguridad

Al tener la completa jurisdicción sobre la base de datos, el DBA (por supuesto, bajo la dirección apropiada del administrador de datos) puede asegurar que el único medio de acceso a la base de datos sea a través de los canales adecuados y por lo tanto puede definir las reglas o **restricciones de seguridad** que serán verificadas siempre que se intente acceder a datos sensibles. Es posible establecer diferentes restricciones para cada tipo de acceso (recuperación, inserción, eliminación, etcétera) para cada parte de la información de la base de datos. Sin embargo, observe que sin dichas restricciones la seguridad de los datos podría de hecho estar en mayor riesgo que en un sistema de archivos tradicionales (dispersos); es decir, la naturaleza centralizada de un sistema de base de datos *requiere,* en cierto sentido, que también sea establecido un buen sistema de seguridad.

Es posible equilibrar los requerimientos en conflicto

Al conocer los requerimientos generales de la empresa (a diferencia de los requerimientos de los usuarios individuales), el DBA puede estructurar los sistemas de manera que ofrezcan un servicio general que sea "el mejor para la empresa" (de nuevo bajo la dirección del administrador de datos). Por ejemplo, es posible elegir una representación física de los datos

* Por otra parte, los sistemas de un solo usuario a menudo no proporcionan ningún tipo de manejo de transacciones sino que simplemente dejan el problema al usuario.

almacenados que proporcione un acceso rápido para las aplicaciones más importantes (posiblemente a costa de un acceso más lento para otras aplicaciones).

Es posible hacer cumplir los estándares

Con el control central de la base de datos, el DBA (una vez más, bajo la dirección del administrador de datos) puede asegurar que todos los estándares aplicables en la representación de los datos sean observados. Estos estándares podrían incluir alguno o todos los siguientes: departamentales, de instalación, corporativos, de la industria, nacionales e internacionales. Es conveniente estandarizar la representación de datos, en particular como un auxiliar para el *intercambio de datos* o para el movimiento de datos entre sistemas (esta consideración se ha vuelto particularmente importante con el advenimiento de los sistemas distribuidos; vea los capítulos 2 y 20). En forma similar, los estándares en la asignación de nombres y en la documentación de los datos también son muy convenientes como una ayuda para compartir y entender los datos.

Es probable que la mayoría de las ventajas mencionadas arriba sean bastante obvias. Sin embargo, es necesario agregar a la lista un punto que podría no ser tan obvio (aunque de hecho está implícito en otros); se trata de *dar independencia a los datos.* (Estrictamente hablando, éste es un *objetivo* de los sistemas de bases de datos, en vez de una ventaja). El concepto de la independencia de los datos es tan importante que le dedicamos una sección aparte.

1.5 LA INDEPENDENCIA DE LOS DATOS

Comenzaremos por observar que existen dos clases de independencia de los datos, física y lógica [1.3-1.4]; sin embargo, por el momento nos concentraremos sólo en la clase física. Por lo tanto, mientras no se diga otra cosa, el término no calificado "independencia de datos" deberá entenderse específicamente como independencia *física* de los datos. En los capítulos 2, 3 y en especial en el 9, abordaremos la independencia lógica de los datos. *Nota:* Tal vez también debamos decir que el término "independencia de los datos" no es muy adecuado (no capta muy bien la naturaleza de lo que en realidad está sucediendo); sin embargo, es el término utilizado tradicionalmente y nos apegaremos a él en este libro.

Podemos entender más fácilmente la independencia de los datos considerando a su opuesto. Las aplicaciones implementadas en sistemas más antiguos (los sistemas anteriores a los relacionales o incluso anteriores a las bases de datos) tienden a ser *dependientes de los datos. Esto* significa que la forma en que físicamente son representados los datos en el almacenamiento secundario y la técnica empleada para su acceso, son dictadas por los requerimientos de la aplicación en consideración, y más aún, significa que *el conocimiento de esa representación física y esa técnica de acceso están integrados dentro del código de la aplicación.*

- *Ejemplo:* Suponga que tenemos una aplicación que utiliza el archivo EMPLEADO de la figura 1.5 y suponga que se decidió, por motivos de rendimiento, que el archivo estaría indexado en su campo "nombre del empleado". En un sistema antiguo, la aplicación en cuestión generalmente estaría al tanto del hecho de que existe el índice, así como de la secuencia de registros que define ese índice; y la estructura de la aplicación estaría construida alrededor de ese conocimiento. En particular, la forma exacta de los diversos accesos a datos y rutinas de verificación de excepciones dentro de la aplicación, dependerá en gran

medida de los detalles de la interfaz que el software de administración de datos presenta a la aplicación.

Decimos que una aplicación como la del ejemplo es **dependiente de los datos,** debido a que es imposible modificar la representación física (la forma en que los datos están físicamente representados en el almacenamiento) o la técnica de acceso (la forma en que son accedidos físicamente) sin afectar a la aplicación de manera drástica. Por ejemplo, no sería posible reemplazar el índice del ejemplo por un esquema de dispersión sin hacer modificaciones mayores a la aplicación. Lo que es más, las partes de la aplicación que requieren de alteración en dicha situación son precisamente las partes que se comunican con el software de administración de datos; las dificultades implicadas son irrelevantes para el problema que originalmente debería resolver la aplicación; es decir, son dificultades *presentadas* por la naturaleza de la interfaz de administración de datos.

Sin embargo, en un sistema de base de datos, sería en extremo inconveniente permitir que las aplicaciones fuesen dependientes de los datos en el sentido descrito; por lo menos por las dos razones siguientes:

1. Las distintas aplicaciones requerirán visiones diferentes de los mismos datos. Por ejemplo, suponga que antes de que la empresa introduzca su base de datos integrada hay dos aplica ciones A y B que poseen cada una un archivo privado con el campo "saldo del cliente". Sin embargo, suponga que la aplicación A almacena dicho campo en formato decimal, mien tras que la aplicación B lo almacena en binario. Aún sería posible integrar los dos archivos y eliminar la redundancia, suponiendo que el DBMS esté listo y sea capaz de realizar todas las conversiones necesarias entre la representación almacenada elegida (la cual podría ser decimal o binaria, o quizá alguna otra) y la forma en que la aplicación desea verla. Por ejem plo, si se decide almacenar el campo en decimal, entonces todo acceso por parte de B re querirá una conversión hacia o desde el formato binario.

   Éste es un ejemplo muy trivial del tipo de diferencias que podrían existir en un sistema de base de datos entre los datos como los ve una aplicación dada y los datos como están almacenados físicamente. Más adelante en esta sección, consideraremos muchas otras posibles diferencias.

2. El DBA debe tener la libertad de cambiar las representaciones físicas o la técnica de acceso en respuesta a los requerimientos cambiantes, sin tener que modificar las aplicaciones exis tentes. Por ejemplo, es posible incorporar nuevos tipos de datos a la base de datos, adoptar nuevos estándares, cambiar las prioridades (y por lo tanto los requerimientos de rendimiento relativo), tener nuevos dispositivos disponibles, etcétera. Si las aplicaciones son dependien tes de los datos, estos cambios necesitarán por lo regular cambios correspondientes en los programas, ocupando así un esfuerzo de programación que de otro modo estaría disponible para la creación de nuevas aplicaciones. Aun en la actualidad, es muy común encontrar que una parte importante del esfuerzo de programación disponible, está dedicada a este tipo de mantenimiento (¡imagine el "problema del año 2000"!), lo cual por supuesto no es el mejor uso de un recurso escaso y valioso.

De aquí que dar independencia a los datos sea un objetivo principal de los sistemas de base de datos. Podemos definir la independencia de los datos como **la inmunidad de las aplicaciones a cambios en la representación física y en la técnica de acceso;** lo que implica desde luego que las aplicaciones involucradas no dependan de ninguna representación física o técnica de acceso

en particular. En el capítulo 2 describimos la arquitectura de los sistemas de bases de datos que proporciona el fundamento para lograr este objetivo. Sin embargo, antes de eso consideremos con mayor detalle algunos ejemplos de los tipos de cambios que el DBA desearía hacer y ante los cuales, por lo tanto, quisiéramos que las aplicaciones fuesen inmunes.

Comenzaremos por definir tres términos: *campo almacenado, registro almacenado* y *archivo almacenado* (consulte la figura 1.7).

Un **campo almacenado** es, en general, la unidad más pequeña de datos almacenados. La base de datos contendrá muchas **ocurrencias (o ejemplares)** de los diversos **tipos** de campos almacenados. Por ejemplo, una base de datos que contiene información sobre los diferentes tipos de partes podría incluir un tipo de campo almacenado con el nombre "número de parte" y luego podría existir una ocurrencia de ese campo almacenado para cada tipo de parte (tornillo, bisagra, tapa, etcétera).

Nota: En la práctica es común omitir los calificadores "tipo" y "ocurrencia" y depender del contexto para indicar a cuál se hace referencia. Aunque existe un ligero riesgo de

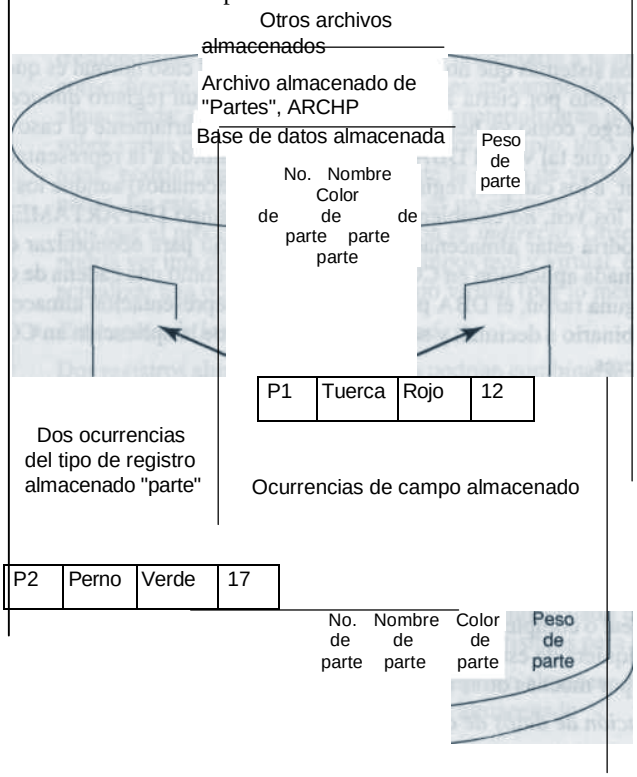


Figura 1.7 Archivos, registros y campos almacenados.

confusión, esta práctica es conveniente y nosotros mismos la adoptaremos de vez en cuando en el libro. (Esta observación se aplica también a los registros almacenados. Vea el siguiente párrafo.)

- Un **registro almacenado** es un conjunto de campos almacenados relacionados. Una vez más distinguimos entre tipo y ocurrencia. Una **ocurrencia (o ejemplar)** de registro almacenado consta de un grupo de ocurrencias de campos almacenados relacionados. Por ejemplo, una ocurrencia de registro almacenado dentro de la base de datos "partes" podría consistir en una ocurrencia de cada uno de los siguientes campos almacenados: número de parte, nombre de parte, color de parte y peso de parte. Decimos que la base de datos contiene muchas ocurrencias del **tipo** de registro almacenado "parte" (una vez más, una ocurrencia por cada clase de parte).
- Por último, un **archivo almacenado** es la colección de todas las ocurrencias existentes actualmente para un tipo de registro almacenado. *Nota:* Por simplicidad damos por hecho que todo archivo almacenado contiene sólo un tipo de registro almacenado. Esta simplificación no afecta sustancialmente ninguna de las explicaciones subsecuentes.

Ahora, en los sistemas que no son de bases de datos, el caso normal es que cualquier registro *lógico* dado (visto por cierta aplicación) es idéntico a un registro *almacenado* correspondiente. Sin embargo, como ya hemos visto éste no es necesariamente el caso en un sistema de base de datos, ya que tal vez el DBA necesita hacer cambios a la representación almacenada de datos (es decir, a los campos, registros y archivos almacenados) aunque los datos, tal y como las aplicaciones los ven, *no* cambien. Por ejemplo, el campo DEPARTAMENTO del archivo EMPLEADO podría estar almacenado en formato binario para economizar espacio, mientras que una determinada aplicación en COBOL podría verlo como una cadena de caracteres. Y más adelante, por alguna razón, el DBA podría modificar la representación almacenada de ese campo, digamos de binario a decimal, y seguir permitiendo que la aplicación en COBOL lo viese en forma de caracteres.

Como mencioné anteriormente, una diferencia como ésta, que comprende una conversión del tipo de datos de cierto campo en cada acceso, es comparativamente menor. No obstante, generalmente la diferencia entre lo que la aplicación ve y lo que en realidad está almacenado podría ser muy considerable. Para ampliar esta observación, presentamos a continuación una lista de los aspectos de la representación almacenada que podrían estar sujetos a cambio. En cada caso. usted deberá considerar lo que el DBMS tendría que hacer para que las aplicaciones sean inmunes a dicho cambio (si es que siempre puede lograrse esa inmunidad).

- *Representación de datos numéricos*

  Un campo numérico podría estar almacenado en la forma aritmética interna (por ejemplo, decimal empacado) o como una cadena de caracteres. En ambas formas, el DBA debe elegir una base apropiada (por ejemplo, binaria o decimal), una escala (de punto fijo o flotante), un modo (real o complejo) y una precisión (el número de dígitos). Podría ser necesario modificar cualquiera de estos aspectos para mejorar el rendimiento, para apegarse a un nuevo estándar o por muchas otras razones.

- *Representación de datos de caracteres*

  Un campo de cadena de caracteres podría ser almacenado mediante cualquiera de los distintos conjuntos de caracteres codificados (por ejemplo, ASCII, EBCDIC o Unicode).

Unidades para datos numéricos

Las unidades en un campo numérico podrían cambiar (por ejemplo, de pulgadas a centímetros durante un proceso de conversión al sistema métrico decimal).

Codificación de los datos

En ciertas situaciones podría ser conveniente representar los datos almacenados por medio de valores codificados. Por ejemplo, el campo "color de parte", que la aplicación ve como una cadena de caracteres ("Rojo" o "Azul" o "Verde" ...), podría ser almacenado como un solo dígito decimal, interpretado de acuerdo con el esquema de codificación 1 = "Rojo", 2 = "Azul", etcétera.

Materialización de los datos

En la práctica, el campo *lógico* (como lo ve una aplicación) corresponde por lo regular a cierto campo almacenado específico; aunque, como ya hemos visto, podría haber diferencias en el tipo de datos, la codificación, etcétera. En tal caso, el proceso de materialización (es decir, la construcción de una ocurrencia del campo lógico a partir de la ocurrencia correspondiente del campo almacenado y presentarla a la aplicación) podría ser considerado como *directo.* Sin embargo, en ocasiones un campo lógico no tendrá una sola contraparte almacenada; en su lugar, sus valores se materializarán por medio de algún cálculo, tal vez sobre varias ocurrencias almacenadas. Por ejemplo, los valores del campo lógico "cantidad total" podrían materializarse mediante la suma de varias cantidades individuales almacenadas. En este caso, "cantidad total" es un ejemplo de un campo **virtual,** por lo que decimos que el proceso de materialización es *indirecto.* Observe, sin embargo, que el usuario podría ver una diferencia entre los campos real y virtual, en tanto que podría no ser posible actualizar una ocurrencia de un campo virtual (por lo menos no directamente).

Estructura de los registros almacenados

Dos registros almacenados existentes podrían combinarse en uno. Por ejemplo, los registros almacenados

no. de parte	color de parte

no. de parte	peso de parte

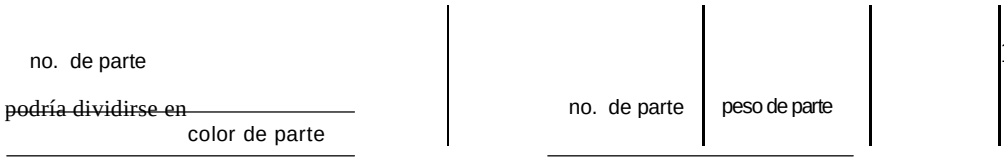
podrían combinarse para formar

no. de parte	color de parte	peso de parte

Un cambio así podría ocurrir cuando las aplicaciones existentes están integradas dentro del sistema de base de datos. Ello implica que el registro lógico de una aplicación podría consistir en un subconjunto propio del registro almacenado correspondiente; es decir, ciertos campos de ese registro almacenado serían invisibles para la aplicación en cuestión.

Como alternativa, un solo tipo de registro almacenado podría ser dividido en dos. Invirtiendo el ejemplo anterior, el registro almacenado

no. de parte	color de parte	peso de parte



1.6

Por ejemplo, esta separación permitiría que las porciones del registro original utilizadas con menos frecuencia sean almacenadas en un dispositivo más lento. Esto implica que un registro lógico de una aplicación podría contener campos de varios registros almacenados distintos; es decir, podría ser un superconjunto propio de cualquiera de esos registros almacenados.

■ *Estructura de los archivos almacenados*

Un determinado archivo almacenado puede ser implementado físicamente en el almacenamiento en una amplia variedad de formas. Por ejemplo, podría estar contenido completamente dentro de un solo volumen de almacenamiento (por ejemplo, un solo disco) o podría estar esparcido en varios volúmenes (posiblemente en diferentes tipos de dispositivos); podría o no tener una secuencia física de acuerdo con los valores de algún campo almacenado; podría o no tener otra secuencia de una o más formas en algún otro medio (digamos, en uno o más índices o en una o más cadenas de apuntadores insertados, o ambos); podría o no ser accesible mediante algún esquema de dispersión; los registros almacenados podrían o no estar bloqueados físicamente; y así sucesivamente. Pero ninguna de estas consideraciones deberá afectar de alguna manera a las aplicaciones (salvo, por supuesto, en el rendimiento).

Esto concluye nuestra lista de aspectos de la representación de datos almacenados que están sujetos a un posible cambio. La lista implica (entre otras cosas) que la base de datos podrá crecer sin dañar las aplicaciones existentes; de hecho, permitir que la base de datos crezca sin dañar de manera lógica las aplicaciones existentes es una de las razones más importantes para requerir, en primer lugar, la independencia de los datos. Por ejemplo, debe ser posible ampliar un registro almacenado existente agregando nuevos campos almacenados, lo que de manera típica representa más información con respecto a algún tipo de entidad existente (por ejemplo, un campo de "costo unitario" podría ser agregado al registro almacenado "parte"). Estos nuevos campos deben simplemente ser invisibles para las aplicaciones existentes. En forma similar, debe ser posible agregar tipos de registros almacenados completamente nuevos (y por lo tanto nuevos archivos almacenados), sin necesidad de algún cambio a las aplicaciones existentes; generalmente, tales registros representarían nuevos tipos de entidad (por ejemplo, un tipo de registro "proveedor" podría ser agregado a la base de datos "partes"). De nuevo, dichas adiciones deberán ser invisibles para las aplicaciones existentes.

A esta altura, quizás ya se ha dado cuenta de que la independencia de los datos es una de las razones de por qué es tan importante separar el modelo de datos de su implementación (como se explicó casi al final de la sección 1.3): En el grado en que no hagamos esa separación, no lograremos la independencia de los datos. Por lo tanto, la falla general al no realizar apropiadamente dicha separación, en especial en los sistemas SQL actuales, es particularmente angustiante. *Nota:* Con estas observaciones no pretendemos decir que los sistemas SQL actuales no proporcionen en absoluto la independencia de los datos, sólo queremos indicar que proporcionan mucho menos de lo que en teoría son capaces los sistemas relaciónales. En otras palabras, la independencia de los datos no es algo absoluto (diferentes sistemas proporcionan distintos grados de independencia y algunos, si los hay, no ofrecen ninguna); los sistemas SQL ofrecen más que los sistemas antiguos, pero aún no son perfectos, como veremos en los capítulos que vienen.

1.6 LOS SISTEMAS RELACIÓNALES Y OTROS SISTEMAS

Como mencioné al final de la sección 1.3, los productos DBMS que se basan en *el modelo de datos relacional* (los "sistemas relaciónales") han venido a dominar el mercado de las bases de datos. Lo que es más, la mayor parte de la investigación sobre bases de datos en los últimos 30 años, se ha basado (aunque en algunos casos un poco en forma indirecta) en este modelo. De hecho, la presentación del modelo relacional en 1969-70 fue de manera innegable el evento más importante en toda la historia de las bases de datos. Por estas razones —más la razón de que el modelo relacional está sólidamente fundamentado en la lógica y en las matemáticas y por lo tanto ofrece un vehículo ideal para la enseñanza de los principios de las bases de datos— el énfasis de este libro (como señalé en la sección 1.1) descansa en gran medida en los sistemas relaciónales.

¿Qué es exactamente un sistema relacional? Es obvio en este punto tan inicial del libro, que no es posible contestar esta pregunta por completo; pero es posible e incluso conveniente dar una respuesta ordinaria y pronta, la cual podremos detallar más adelante. Brevemente, y muy vagamente, un sistema relacional es aquél en el que:

'1. Los datos son percibidos por el usuario como tablas (y nada más que tablas); y

'2. Los operadores disponibles para el usuario (por ejemplo, para recuperación) son operadores que generan nuevas tablas a partir de las anteriores. Por ejemplo, hay un operador *restringir* que extrae un subconjunto de filas de una tabla dada y otro operador proyector que extrae un subconjunto de columnas; y por supuesto, un subconjunto de filas y un subconjunto de columnas de una tabla pueden de por sí ser vistos como tablas, como veremos en un momento.

Nota: La razón por la que dichos sistemas se denominan "relaciónales" es que el término *relación* es básicamente el término matemático para *tabla;* de hecho, los términos *relación* y *tabla* pueden tomarse como sinónimos, por lo menos para fines informales (para una mayor explicación, vea los capítulos 3 y 5). Quizá debamos agregar que la razón *no* es en definitiva que el término *relación* sea "básicamente un término matemático para" un *vínculo* en el sentido de los diagramas de entidad/vínculo (vea la sección 1.3); de hecho, existe muy poca conexión directa entre los sistemas relaciónales y dichos diagramas, como veremos en el capítulo 13.

Para repetir, posteriormente detallaremos las definiciones anteriores; pero éstas servirán por el momento. La figura 1.8 ofrece una ilustración. Los datos (ver la parte a. de la figura) consisten en una sola tabla, denominada CAVA (de hecho, ésta es una versión de la tabla CAVA de la figura 1.1, que fue reducida para hacerla más manejable). En la parte b. de la figura se muestran dos recuperaciones de ejemplo; una que comprende una *restricción* u operación del subconjunto de filas y la otra una *proyección* u operación del subconjunto de columnas. *Nota:* Una vez más, las dos recuperaciones están expresadas en SQL.

Ahora podemos distinguir entre los sistemas relaciónales y los no relaciónales de la siguiente manera. Como ya mencionamos, el usuario de un sistema relacional ve tablas y nada más que tablas. En contraste, el usuario de un sistema no relacional ve otras estructuras de datos, ya sea en lugar de las tablas de un sistema relacional o además de ellas. A su vez, esas otras estructuras requieren de otros operadores para manipularlas. Por ejemplo, en un sistema **jerárquico** como el IMS de IBM, los datos son representados ante el usuario como un conjunto de estructuras de árbol (jerarquías), y los operadores que se proporcionan para manipular dichas estructuras in-

a. *Tabla dada:* CAVA

VINO	AÑO	BOTELLAS
Chardonnay	1996	4
Fumé Blanc	1996	2
Pinot Noir	1993	3
Zinfandel	1994	9

b. *Operadores (ejemplos):*

1. *Restringir:* *Resultado:*

```
SELECT VINO, AÑO, BOTELLAS
FROM CAVA WHERE AÑO > 1995
;
```

VINO	AÑO	BOTELLAS
Chardonnay	1996	4
Fumé Blanc	1996	2

2. *Proyectar:* *Resultado:*

```
SELECT VINO, BOTELLAS
FROM  CAVA ;
```

VINO	BOTELLAS
Chardonnay	4
Fumé Blanc	2
Pinot Noir	3
Zinfandel	9

Figura 1.8 Estructura de datos y operadores en un sistema relacional (ejemplos).

cluyen operadores para *apuntadores de recorrido; es decir,* los apuntadores que representan las rutas jerárquicas hacia arriba y hacia abajo en los árboles. (En contraste, es una característica distintiva de los sistemas relaciónales el que no contengan, como hemos visto, dichos apuntadores.)

Para llevar este aspecto un poco más lejos: Los sistemas de bases de datos pueden de hecho ser divididos convenientemente en categorías de acuerdo con los operadores y estructuras de datos que presentan al usuario. De acuerdo con este esquema, los sistemas más antiguos (prerrelacionales) se ubican dentro de tres grandes categorías, los sistemas **de listas invertidas, jerárquicos** y **de red.*** En este libro no exponemos con detalle estas categorías debido a que, por lo menos desde un punto de vista tecnológico, deben ser vistos como obsoletos. (Si usted está interesado, puede encontrar descripciones tutoriales de los tres en la referencia [1.5].) Sin embargo, debemos mencionar por lo menos que el término *red* en este contexto no tiene nada que ver con una red de *comunicaciones;* más bien se refiere (para repetir) a las clases de estructuras de datos y operadores que manejan los sistemas en cuestión.

---

*Por analogía con el modelo relacional, las ediciones anteriores de este libro se referían a los *modelos* de listas invertidas, jerárquicos y de red (y gran parte de la literatura aún lo hace). Sin embargo, hablar en esos términos es más bien inexacto, ya que a diferencia del modelo relacional, los "modelos" de listas invertidas, jerárquicos y de red se definieron *después del hecho;* es decir, los productos comerciales de listas invertidas, jerárquicos y de red fueron implementados *primero* y los "modelos" correspondientes fueron definidos *posteriormente* mediante un proceso de inducción —en este contexto, un término formal para adivinar— a partir de las implementaciones existentes.

Nota: En ocasiones, a los sistemas de red se les denomina sistemas **CODASYL** o sistemas **DBTG,** por el grupo que los propuso; es decir, el DBTG (Grupo de Tareas de Bases de Datos) del CODASYL (Congreso sobre Lenguajes de Sistemas de Datos). Probablemente el ejemplo más conocido de estos sistemas es IDMS (de Computer Associates International Inc). Al igual que los sistemas jerárquicos, aunque a diferencia de los relaciónales, todos estos sistemas exponen (entre otras cosas) apuntadores ante el usuario.

Los primeros productos **relaciónales** comenzaron a aparecer a finales de los años setenta y principios de los ochenta. Hasta este momento, la gran mayoría de los sistemas de base de datos son relaciónales y operan prácticamente en todo tipo de plataforma de hardware y de software disponible. Los ejemplos principales comprenden, en orden alfabético, a DB2 (varias versiones) de IBM Corp.; Ingress II de Computer Associates International Inc.; Informix Dynamic Server de Informix Software Inc.; Microsoft SQL Server de Microsoft Corp.; Oracle 8i de Oracle Corp.; y Sybase Adaptive Server de Sybase Inc. *Nota:* Cuando tengamos motivos para referirnos a cualquiera de estos productos en el resto del libro, lo haremos por sus nombres abreviados(como lo hace la mayoría de la industria, de manera informal): DB2, Ingres, Informix, SQL Server, Oracle y Sybase, respectivamente.

En fechas más recientes, se han puesto a disposición ciertos productos de **objetos** y **objeto/reladonales.*** Los sistemas objeto/relaciónales representan en su mayoría extensiones compatibles hacia arriba de algunos de los productos relaciónales originales, como es el caso de DB2 e Informix; los sistemas de objetos (en ocasiones *orientados a objetos)* representan intentos de hacer algo completamente diferente, como es el caso de GemStone de GemStone Systems Inc. y de Versant ODBMS de Versant Object Technology. En la parte VI de este libro, explicaremos estos sistemas más recientes.

Además de los distintos enfoques antes mencionados, a lo largo de los años las investigaciones han seguido una variedad de esquemas alternativos, incluyendo los enfoques **multidimensional** y el **basado en la lógica** (también llamado *deductivo* o *experto).* En el capítulo 21 explicaremos los sistemas multidimensionales y en el capítulo 23 los sistemas basados en la lógica.

1.7 RESUMEN

Cerramos este capítulo introductorio con un resumen de las ideas principales expuestas. Primero, es posible pensar en un **sistema de base de datos** como un sistema de registros computarizado. Dicho sistema comprende a los propios **datos** (almacenados en la **base de datos),** al **hardware,** al **software** (en particular al **sistema de administración de base de datos** o DBMS) y (¡lo más importante!) a los **usuarios.** A su vez, los usuarios pueden ser divididos en **programadores de aplicaciones, usuarios finales** y **administrador de base de datos** o DBA. El DBA es el responsable de administrar la base de datos y el sistema de base de datos, de acuerdo con las políticas establecidas por el **administrador de datos.**

Las bases de datos están **integradas** y por lo regular son **compartidas;** se emplean para almacenar **datos persistentes.** Dichos datos pueden considerarse, de manera útil aunque informal, como una representación de **entidades,** junto con los **vínculos** que están entre éstas (aunque de hecho, un vínculo es en realidad sólo una clase especial de entidad). Analizaremos brevemente la idea de los **diagramas de entidad/vínculo.**

---

*Aquí el término *objeto* tiene un significado más bien específico, el cual explicaremos en la parte VI. Antes de ese punto, usaremos el término en su sentido general, salvo que se indique lo contrario.

Los sistemas de bases de datos ofrecen diversos beneficios. Uno de los más importantes es el de la **independencia** (física) **de los datos.** Podemos definir la independencia de los datos como la inmunidad que tienen los programas de aplicación ante los cambios en la forma almacenar o acceder físicamente a los datos. Entre otras cosas, la independencia de los datos requiere que se haga una clara distinción entre el **modelo de datos y** su **implementación.** (De paso, le recordamos que el término *modelo de datos,* quizás en forma desafortunada, tiene dos significados diferentes.)

Los sistemas de bases de datos también soportan por lo regular **transacciones** o unidades de trabajo lógicas. Una ventaja de las transacciones es que está garantizado que sean **atómicas** (todo o nada), incluso si el sistema falla a mitad de su ejecución.

Por último, los sistemas de bases de datos pueden estar fundamentados en varias teorías diferentes. En particular, los sistemas relaciónales se basan en una teoría formal denominada **modelo relacional,** según la cual los datos están representados como filas de tablas (interpretadas como **proposiciones verdaderas)** y cuentan con operadores que manejan directamente el proceso de **inferir** proposiciones verdaderas adicionales a partir de las ya dadas. Desde una perspectiva tanto económica como teórica, los sistemas relacionales son sin duda los más importantes (y no es probable que esta situación cambie en el futuro previsible). Vimos algunos ejemplos de SQL, el lenguaje estándar para tratar con los sistemas relacionales (en particular, ejemplos de las instrucciones **SELECT, INSERT, UPDATE y DELETE** de SQL). Este libro se basará en gran medida en los sistemas relacionales, aunque por las razones que expuse en el prefacio, no demasiado en SQL *per se.*

EJERCICIOS

1.1 Defina los siguientes términos:

acceso concurrente
administración de datos
aplicación en línea
archivo almacenado
base de datos
campo almacenado
compartir
datos persistentes
DBA
DBMS
diagrama de entidad/vínculo
entidad
independencia de los datos
integración
integridad
interfaz controlada por comandos
interfaz controlada por formularios
interfaz controlada por menus
lenguaje de consulta
propiedad
redundancia
registro almacenado
seguridad
sistema de base de datos
sistema multiusuario
transacción
vínculo
vínculo binario

1.2 ¿Cuáles son las ventajas de usar un sistema de base de datos?

1.3 ¿Cuáles son las desventajas de usar un sistema de base de datos?

1.4 ¿Qué entiende por el término *sistema relacionan* Distinga entre los sistemas relacionales y los no relacionales.

1.5 ¿Qué entiende por el término *modelo de datos?* Explique la diferencia entre un modelo de datos y su implementación. ¿Por qué es importante la diferencia?

1.6 Muestre los efectos que tienen las siguientes operaciones SQL de recuperación sobre la base de datos de la cava de vinos mostrada en la figura 1.1.

a.
```
SELECT VINO, PRODUCTOR
FROM   CAVA
WHERE NICHO# • 72 ;
```

b.
```
SELECT VINO, PRODUCTOR
FROM   CAVA
WHERE   AÑO > 1996  ;
```

C.
```
SELECT NICHO#,  VINO,  AÑO
FROM    CAVA WHERE    LISTO <
1999  ;
```

d.
```
SELECT VINO,  NICHO#,  AÑO
FROM    CAVA
WHERE   PRODUCTOR =  ' Robt.  Mondavi'
AND     BOTELLAS > 6  ;
```

1.7 A partir de cada una de sus respuestas al ejercicio 1.6, dé en sus propias palabras una interpre tación (como una proposición verdadera) de una fila típica.

1.8 Muestre los efectos de las siguientes operaciones SQL de actualización sobre la base de datos de la cava de vinos de la figura 1.1.

a.
```
INSERT
INTO  CAVA ( NICH0#, VINO, PRODUCTOR, AÑO, BOTELLAS, LISTO )
VALUES ( 80, 'Syrah', 'Meridian', 1994, 12, 1999 ) ;
```

b.
```
DELETE
FROM  CAVA
WHERE LISTO > 2000 ;
```

C.
```
UPDATE CAVA
SET   BOTELLAS = 5
WHERE NICHO# = 50 ;
```

d.
```
UPDATE CAVA
SET   BOTELLAS = BOTELLAS + 2
WHERE  NICHO# = 50 ;
```

1.9 Escriba instrucciones SQL para realizar las siguientes operaciones en la base de datos de la cava de vinos:

a. Obtenga el número de nicho, el nombre del vino y el número de botellas de todos los vinos Geyser Peak.
b. Obtenga el número de nicho y el nombre de todos los vinos que tengan en existencia más de cinco botellas.
c. Obtenga el número de nicho de todos los vinos rojos.
d. Agregue tres botellas al nicho número 30.
e. Elimine de las existencias todo el Chardonnay.
f. Agregue una entrada para un nuevo caso (12 botellas) de Gary Farrell Merlot: nicho número 55, año 1996, listo en el 2001.

1.10 Suponga que tiene una colección de música clásica que consta de CDs, LPs y cintas de audio y desea elaborar una base de datos que le permita determinar qué grabaciones posee de un compositor específico (por ejemplo, Sibelius), director (por ejemplo, Simon Rattle), solista (por ejemplo, Arthur Grumiaux), obra (por ejemplo, la quinta sinfonía de Beethoven), orquesta (por ejemplo, la Orquesta filarmónica de la ciudad de Nueva York), tipo de obra (por ejemplo, concierto para violin) o grupo de cámara (por ejemplo, Kronos Quartet). Dibuje un diagrama de entidad/vínculo para esta base de datos (como el de la figura 1.6).

REFERENCIAS Y BIBLIOGRAFÍA

1.1 E. F. Codd: "Data Models in Database Management", Proc. Wokshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colo, (junio, 1980); *ACM SIGART Newsletter No.* 74 (enero, 1981); *ACM SIGMODRecord 11*,No. 2(febrero, 1981);*ACM SIGPLAN Notices 16,No.l* (ene ro, 1981).

Codd fue el inventor del modelo relacional, el cual describió primero en la referencia [5.1]. Sin embargo, dicha referencia ¡no define el término *modelo de datos* como tal!; aunque el presente artículo (muy posterior), sí lo hace. Éste aborda la pregunta ¿A qué fines pretenden servir los modelos de datos en general y el modelo relacional en particular? Después continúa ofreciendo evidencia para apoyar la afirmación de que, al contrario de la creencia popular, el modelo relacional fue de hecho el primer modelo de datos en ser definido. (En otras palabras, Codd afirma en cierto modo ser el inventor del concepto de modelo de datos en general, así como del modelo de datos relacional en particular.)

1.2 Hugh Darwen: "What a Database *Really Is:* Predicates and Propositions", en C. J. Date, Hugh Darwen y David McGoveran, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).

Este artículo ofrece una explicación informal (pero precisa) de la idea, expuesta brevemente al final de la sección 1.3, de que es posible visualizar a una base de datos como un conjunto de proposiciones verdaderas.

1.3 C. J. Date y P. Hopewell: "Storage Structures and Physical Data Independence", Proc. 1971 ACM SIGFIDET Workshop on Data Definition, Access, and Control, San Diego, California (noviembre, 1971).

1.4 C. J. Date y P. Hopewell: "File Definition and Logical Data Independence", Proc. 1971 ACM SIG FIDET Workshop on Data Definition, Access, and Control, San Diego, California (noviembre, 1971).

Las referencias [1.3-1.4] fueron los primeros artículos en definir y distinguir entre la independencia física y lógica de los datos.

1.5 C. J. Date: *Relational Database Writings 1991-1994.* Reading, Mass.: Addison-Wesley (1995).

RESPUESTAS A EJERCICIOS SELECCIONADOS

1.3 Algunas desventajas son las siguientes:

- Podría comprometerse la seguridad (sin los controles adecuados);
- Podría comprometerse la integridad (sin los controles adecuados);
- Podría requerirse de hardware adicional;
- La sobrecarga en el rendimiento podría ser importante;

La operación exitosa es crucial (la empresa podría ser altamente vulnerable a fallas);

Es probable que el sistema sea complejo (aunque dicha complejidad debe ocultarse ante el usuario).

1.6 a.

VINO	PRODUCTOR
Zinfandel	Rafanelli

b.

VINO	PRODUCTOR
Chardonnay Chardonnay Joh. Riesling Fumé Blanc Gewurztraminer	Buena Vista Geyser Peak Jekel Ch. St. Jean Ch. St. Jean

c.

NICHO#	VINO	AÑO
6 22 52	Chardonnay Fumé Blanc Pinot Noir	1996 1996 1995

d.

VINO	NICH0#	AÑO
Cab. Sauvignon	48	1993

1.7 Sólo presentamos la solución de la parte a.: "Rafanelli es un productor de Zinfandel"; o con más precisión, "Cierto nicho contiene algunas botellas de Zinfandel que produjo Rafanelli en cierto año y estarán listas para su consumo en algún año".

1.8 a. Se agrega una fila a la tabla CAVA para el nicho número 80.

b. Se eliminan de la tabla CAVA las filas de los nichos 45, 48, 64 y 72.

c. Se asigna 5 al número de botellas del nicho número 50.

d. Igual que c.

Por cierto, observe lo conveniente que resulta poder referirse a las filas por el valor de su clave primaria (la clave primaria de la tabla CAVA es {NICHO#}). Vea el capítulo 8.

```
1.9  a. SELECT NICHO#, VINO, BOTELLAS
        FROM  CAVA
        WHERE PRODUCTOR ■ 'Geyser Peak' ;

     b. SELECT NICHO*, VINO
        FROM  CAVA WHERE
        BOTELLAS > 5 ;

     C. SELECT NICHO#
        FROM  CAVA
        WHERE VINO = 'Cab. Sauvignon'
        OR    VINO = 'Pinot Noir'
        OR    VINO = 'Zinfandel'
        OR    VINO ■ 'Syrah'
        OR   ..... ;
```

No hay una respuesta abreviada para esta pregunta, debido a que el "color del vino" no está registrado de manera explícita en la base de datos; por lo tanto, el DBMS no sabe que (por ejemplo) Pinot Noir es rojo.

d.
```
UPDATE CAVA
SET   BOTELLAS = BOTELLAS + 3
WHERE  NICHO# « 30 ;
```

e.
```
DELETE
FROM  CAVA
WHERE VINO - 'Chardonnay' ;
```

f.
```
INSERT
INTO  CAVA ( NICHO#, VINO, PRODUCTOR, AÑO, BOTELLAS, LISTO )
VALUES ( 55, 'Merlot', 'Gary Farrell', 1996, 12, 2001 ) ;
```

CAPITULO 2

Arquitectura de los sistemas de bases de datos

2.1 INTRODUCCIÓN

Ahora estamos en condiciones de presentar la arquitectura para un sistema de base de datos. Nuestro objetivo al presentar esta arquitectura es ofrecer una infraestructura en la que puedan basarse los capítulos siguientes. Dicha infraestructura resulta útil para describir los conceptos generales de las bases de datos y para explicar la estructura de sistemas de bases de datos específicos; pero no afirmamos que todo sistema pueda coincidir enteramente con esta infraestructura en particular, ni queremos sugerir que esta arquitectura represente la única infraestructura posible. En particular, es probable que los sistemas "pequeños" (vea el capítulo 1) no manejen todos los aspectos de la arquitectura. Sin embargo, la arquitectura parece ajustarse bastante bien a la mayoría de los sistemas; es más, es prácticamente idéntica a la arquitectura propuesta por el Grupo de Estudio en Sistemas de Administración de Bases de Datos de ANSI/SPARC (la tan mencionada arquitectura ANSI/SPARC. Vea las referencias [2.1-2.2]). Sin embargo, nosotros decidimos no seguir la terminología ANSI/SPARC en todos sus detalles.

Nota: Este capítulo se asemeja al capítulo 1 en el sentido de que también es en cierto modo abstracto y árido, aunque es fundamental entender el material que contiene para una apreciación completa de la estructura y posibilidades de un sistema de base de datos moderno. Por lo tanto, al igual que en el capítulo 1, tal vez prefiera por ahora sólo darle una leída "ligera" y regresar a él más tarde, cuando sea directamente relevante para los temas que esté abordando.

2.2 LOS TRES NIVELES DE LA ARQUITECTURA

La arquitectura ANSI/SPARC se divide en tres niveles, conocidos como interno, conceptual y externo, respectivamente (vea la figura 2.1). Hablando en términos generales:

- El **nivel interno** (también conocido como el nivel *físico)* es el que está más cerca del al macenamiento físico; es decir, es el que tiene que ver con la forma en que los datos están almacenados físicamente.
- El **nivel externo** (también conocido como el nivel *lógico de usuario)* es el más próximo a los usuarios; es decir, el que tiene que ver con la forma en que los usuarios individuales ven los datos.
- El **nivel conceptual** (también conocido como el nivel *lógico de la comunidad,* o en oca siones sólo como el nivel *lógico,* sin calificar) es un nivel de indirección entre los otros dos.

Observe que el nivel externo tiene que ver con las percepciones de usuarios *individuales,* mientras que el nivel conceptual tiene que ver con la percepción de una *comunidad* de usuarios.

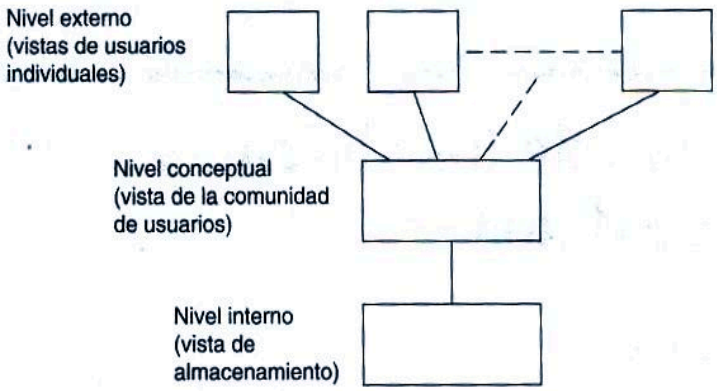


Figura 2.1 Los tres niveles de la arquitectura.

En otras palabras, habrá muchas "vistas externas" distintas, cada una consistente en una representación más o menos abstracta de alguna parte de la base de datos total, y habrá precisamente una "vista conceptual" que del mismo modo consiste en una representación abstracta de la base de datos en su totalidad.* (Recuerde que la mayoría de los usuarios no se interesarán en toda la base de datos, sino sólo en una parte limitada de la misma). En forma similar, habrá precisamente una "vista interna" que represente a la base de datos tal como está almacenada físicamente.

Un ejemplo hará más claras estas ideas. La figura 2.2 muestra la vista conceptual, la vista interna correspondiente y dos de las vistas externas (una para un usuario de PL/I y otra para un usuario de COBOL), todas ellas para una base de datos de personal sencilla. Por supuesto, el ejemplo es completamente hipotético —no pretende reflejar ningún sistema real— y hemos omitido deliberadamente muchos detalles irrelevantes. *Explicación*:

- En el nivel conceptual, la base de datos contiene información concerniente a un tipo de entidad denominada EMPLEADO. Cada empleado individual tiene un NUMERO_EMPLEADO (de seis caracteres), un NUMERO_DEPARTAMENTO (de cuatro caracteres) y un SALARIO (de cinco dígitos decimales).
- En el nivel interno, los empleados están representados por un tipo de registro denominado EMP_ALMACENADO, de veinte bytes de longitud. EMP_ALMACENADO contiene cuatro campos almacenados: un prefijo de seis bytes (que presumiblemente contiene información de control como los indicadores o los apuntadores) y tres campos de datos correspondientes a las tres propiedades de los empleados. Además, los registros de EMP_ALMACENADO están indexados sobre el campo EMP# por medio de un índice de nombre EMPX, cuya definición no se muestra.
- El usuario de PL/I tiene una vista externa de la base de datos en la que cada empleado está representado por un registro PL/I que contiene dos campos (los números de departamento

*Aquí, por *abstracto* simplemente queremos decir que la representación en cuestión comprende construcciones como los registros y campos que están más orientadas a los usuarios, a diferencia de las construcciones como los bits o los bytes que están más orientadas a la máquina.

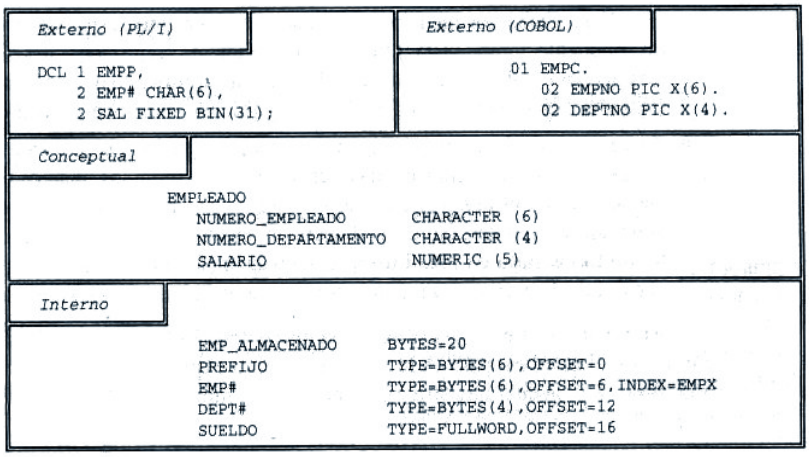


Figura 2.2 Un ejemplo de los tres niveles.

no son de interés para este usuario y por lo tanto fueron omitidos). El tipo del registro está definido por una declaración de estructura PL/I ordinaria según las reglas normales de PL/I.

- En forma similar, el usuario de COBOL tiene una vista externa en la que cada empleado está representado por un registro de COBOL, que una vez más contiene dos campos (esta vez fueron omitidos los salarios). El tipo de registro está definido por una descripción ordinaria de registro COBOL, de acuerdo con las reglas normales de COBOL.

Observe que los elementos correspondientes de los datos pueden tener diferentes nombres en distintos puntos. Por ejemplo, el número de empleado se denomina EMP# en la vista externa de PL/I, EMPNO en la vista externa de COBOL, NUMERO_EMPLEADO en la vista conceptual y nuevamente EMP# en la vista interna. Desde luego, el sistema debe estar al tanto de las correspondencias; por ejemplo, debemos indicar que el campo EMPNO de COBOL se deriva del campo conceptual NUMERO_EMPLEADO, el cual se deriva a su vez del campo almacenado EMP# al nivel interno. Dichas correspondencias, o **transformaciones**, no se muestran de manera explícita en la figura 2.2; para una explicación más amplia, consulte la sección 2.6.

Ahora bien, para los fines del presente capítulo, no tiene mayor importancia si el sistema a considerar es relacional o de otro tipo. Sin embargo, sería útil indicar cómo son concebidos típicamente los tres niveles de la arquitectura en un sistema relacional en particular:

- Primero, el nivel conceptual en dicho sistema será definitivamente relacional, en el sentido de que los objetos visibles en ese nivel serán tablas relacionales y los operadores serán de tipo relacional (incluyendo los operadores *restringir* y *proyectar* en particular, que expusimos brevemente en el capítulo 1).
- Segundo, una vista externa dada también será casi siempre relacional, o algo muy parecido a ello; por ejemplo, las declaraciones de registro de PL/I y COBOL de la figura 2.2 podrían ser consideradas a grandes rasgos como análogas de las declaraciones de PL/I y COBOL, respectivamente, de una tabla relacional en un sistema relacional.

Nota: Debemos mencionar de paso que el término "vista externa" (a menudo abreviado solamente como "vista") tiene por desgracia un significado más bien específico en contextos relacionales y que éste *no* es idéntico al significado que se le asigna en este capítulo. Para una explicación y exposición del significado relacional, consulte los capítulos 3 y 9.

- Tercero, el nivel interno *no* será relacional, ya que los objetos en ese nivel no serán sólo tablas relacionales (almacenadas); en vez de ello, serán los mismos tipos de objetos que se encuentran en el nivel interno de cualquier otro tipo de sistema (registros almacenados, apuntadores, índices, tablas de dispersión, etcétera). De hecho, el modelo relacional como tal **no tiene nada en absoluto que decir** acerca del nivel interno; para repetir lo dicho en el capítulo 1, tiene que ver con la forma en que la base de datos se presenta ante el *usuario.*

Ahora procederemos a explicar con más detalle los tres niveles de la arquitectura, comenzando con el nivel externo. A lo largo de nuestra explicación haremos constantes referencias a la figura 2.3, la cual muestra los principales componentes de la arquitectura y sus interrelaciones.

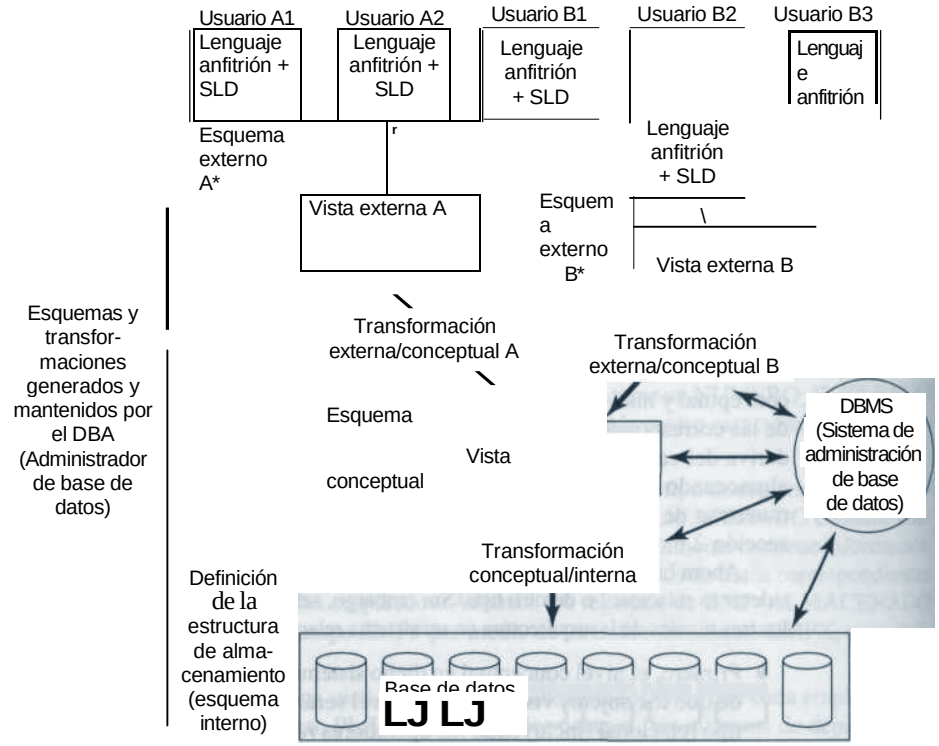


Figura 2.3 Arquitectura detallada del sistema.

2.3 /EL NIVEL EXTERNO

El nivel externo es el nivel del usuario individual. Como expliqué en el capítulo 1, un usuario dado puede ser un programador de aplicaciones o bien un usuario final con cualquier grado de sofisticación. (El DBA es un importante caso especial; pero a diferencia de otros usuarios, el DBA también necesitará interesarse en los niveles conceptual e interno. Vea las dos secciones siguientes.) Cada usuario tiene a su disposición un **lenguaje:**

- Para el programador de aplicaciones, éste será ya sea un lenguaje de programación con vencional (por ejemplo, PL/I, C++, Java) o bien un lenguaje de tipo propietario que sea es pecífico al sistema en cuestión. A menudo, a estos lenguajes de tipo propietario se les denomina lenguajes de "cuarta generación" (4GLs); siguiendo las —¡confusas!— bases de que (a) el código de máquina, el lenguaje ensamblador y los lenguajes como PL/I pueden ser vistos como tres "generaciones" de lenguajes anteriores, y (b) los lenguajes de tipo pro pietario representan la misma clase de mejora sobre los lenguajes de "tercera generación" (3GLs) que la que tuvieron estos lenguajes sobre el lenguaje ensamblador y la que tuvo el lenguaje ensamblador sobre el código de máquina.
- Para el usuario final, el lenguaje será ya sea un lenguaje de consulta o bien algún lenguaje de finalidad específica, tal vez controlado por formularios o por menús, confeccionado para los requerimientos de ese usuario y manejado por algún programa de aplicación en línea (como expliqué en el capítulo 1).

En nuestro caso, lo importante acerca de dichos lenguajes es que incluirán un **sublenguaje de datos;** es decir, un subconjunto del lenguaje total que se ocupe específicamente de los objetos y operaciones de la base de datos. Se dice que el sublenguaje de datos (abreviado como SLD en la figura 2.3) está **incrustado** dentro de su **lenguaje anfitrión** correspondiente. El lenguaje anfitrión es el responsable de proporcionar diversas propiedades que no son específicas de la base de datos, como las variables locales, las operaciones de cálculo, la lógica de bifurcación, etcétera. Un sistema determinado podría manejar cualquier cantidad de lenguajes anfitrión y cualquier número de sublenguajes de datos; sin embargo, un sublenguaje de datos específico soportado por casi todos los sistemas actuales es el lenguaje SQL que explicamos brevemente en el capítulo 1. La mayoría de dichos sistemas permiten que SQL sea utilizado de manera *interactiva,* como un lenguaje de consulta independiente, e *incrustado* en otros lenguajes como PL/I o Java (para una explicación más amplia, consulte el capítulo 4).

Ahora bien, aunque para fines de la arquitectura es conveniente distinguir entre el sublenguaje de datos y el lenguaje anfitrión que lo contiene, ambos podrían *no* ser distintos en lo que al usuario concierne; y de hecho, desde el punto de vista del usuario, es probablemente mejor que no lo sean. Si no son distintos, o si difícilmente pueden distinguirse, decimos que están **fuertemente acoplados.** Si son clara y fácilmente separables, decimos que están **débilmente** acoplados. Aunque algunos sistemas comerciales (en especial los orientados a objetos; vea el capítulo 24) sí manejan el acoplamiento fuerte, la mayoría no lo hace; en particular, los sistemas SQL generalmente sólo manejan el acoplamiento débil. (El acoplamiento fuerte ofrece un conjunto más uniforme de propiedades para el usuario, aunque obviamente implica más esfuerzo por parte de los desarrolladores del sistema, un hecho que presumiblemente cuenta para el *statu quo.)*

En principio, cualquier sublenguaje de datos determinado es en realidad una combinación de por lo menos dos lenguajes subordinados: un DDL **(lenguaje de definición de datos),** que

permite la definición o declaración de objetos de base de datos, y un DML **(lenguaje de manipulación de datos),** que permite la manipulación o procesamiento de dichos objetos. Por ejemplo, considere al usuario de PL/I de la figura 2.2 de la sección 2.2. El sublenguaje para ese usuario consiste en aquellas características de PL/I que se usan para comunicarse con el DBMS:

- La parte del DDL consiste en aquellas construcciones declarativas de PL/I necesarias para declarar objetos de base de datos; la propia instrucción DECLARE (DCL), ciertos tipos de datos de PL/I, tal vez extensiones especiales de PL/I para manejar nuevos objetos que no maneja el PL/I existente.
- La parte del DML consiste en aquellas instrucciones ejecutables de PL/I que transfieren in formación hacia y desde la base de datos; de nuevo, que incluyen tal vez nuevas instruccio nes especiales.

Nota: Para ser más precisos, debemos decir que al momento de la publicación de este libro, PL/I no incluía ninguna característica específica de base de datos. En particular, las instrucciones "DML" por lo regular sólo son instrucciones CALL de PL/I que invocan al DBMS (aunque esas instrucciones podrían de alguna forma estar disfrazadas sintácticamente para hacerlas un poco más sencillas para el usuario; vea la explicación de SQL incrustado, en el capítulo 4).

Volviendo a la arquitectura, ya indicamos que un usuario individual se interesará generalmente sólo por alguna parte de toda la base de datos; es más, la vista que el usuario tiene de esa parte normalmente será un poco abstracta al compararla con la forma en que los datos están almacenados físicamente. El término ANSI/SPARC utilizado para la vista de un usuario individual es el de **vista externa.** Por lo tanto, una vista externa es el contenido de una base de datos como lo ve algún usuario en particular (es decir, para ese usuario, la vista externa *es* la base de datos). Por ejemplo, un usuario del Departamento de Personal podría ver a la base de datos como una colección de ocurrencias de los registros de empleado y departamento, y podría desconocer las ocurrencias de los registros de parte y proveedor que ven los usuarios del Departamento de Compras.

Entonces, en general, una vista externa consiste en muchas ocurrencias de cada uno de los tipos de **registro externo** (que *no* es necesariamente lo mismo que un registro almacenado).* El sublenguaje de datos del usuario está definido en términos de registros externos; por ejemplo, una operación de *recuperación* del DML recuperará ocurrencias de registros externos, no ocurrencias de registros almacenados. *Nota:* Ahora podemos ver que el término "registro lógico", empleado ocasionalmente en el capítulo 1, se refería en realidad a un registro externo. De hecho, desde este punto de vista, trataremos de evitar el término "registro lógico".

Cada vista externa está definida por medio de un **esquema externo,** el cual consiste básicamente en definiciones de cada uno de los diversos tipos de registros externos de esa vista (observe una vez más en la figura 2.2, un par de ejemplos sencillos). El esquema externo fue escrito utilizando la parte DDL del sublenguaje de datos del usuario. (De ahí que en ocasiones se haga referencia a ese DDL como *DDL externo.)* Por ejemplo, el tipo de registro externo de empleado podría definirse como un campo de número de empleado con seis caracteres, más un campo de

---

*Aquí, damos por hecho que toda la información está representada a un nivel externo en forma de registros. Sin embargo, algunos sistemas permiten que la información sea representada en otras formas en vez de, o también en forma de, registros. Para que un sistema emplee dichos métodos alternativos, será necesario adecuar las definiciones y explicaciones dadas en esta sección. Se aplican también observaciones similares a los niveles conceptual e interno. La consideración detallada de estos aspectos está más allá del alcance de esta primera parte del libro; para una explicación más amplia, consulte los capítulos 13 (en especial la sección de "Referencias y bibliografía") y 24.

salario con cinco dígitos (decimales) y así sucesivamente. Además, debe haber una definición de la *transformación* entre el esquema externo y el esquema *conceptual* subyacente (vea la siguiente sección). Más adelante, en la sección 2.6, consideraremos dicha transformación.

4 EL NIVEL CONCEPTUAL

La **vista conceptual** es una representación de todo el contenido de la información de la base de datos, de nuevo (al igual que con la vista externa) en una forma un poco abstracta comparada con la forma en la que por lo regular se almacenan los datos físicamente. También será muy diferente de la forma en que cualquier usuario específico ve los datos. En términos generales, la vista conceptual pretende ser una vista de los datos "tal como son", en vez de tal como los usuarios están obligados a verlos debido a las limitaciones (por ejemplo) del lenguaje o el hardware en particular que pudieran utilizar.

La vista conceptual consiste en muchas ocurrencias de varios tipos de **registro conceptual.** Por ejemplo, podría consistir en un conjunto de ocurrencias de los registros de departamento, más un conjunto de ocurrencias de los registros de empleado, más un conjunto de ocurrencias de los registros de proveedor, más un conjunto de ocurrencias de los registros de parte (y así sucesivamente). Por otra parte, un registro conceptual no es necesariamente lo mismo que un registro externo, ni que un registro almacenado.

La vista conceptual está definida por medio del **esquema conceptual,** el cual comprende definiciones de cada uno de los diversos tipos de registros conceptuales (de nuevo, consulte la figura 2.2 para ver un ejemplo sencillo). El esquema conceptual está escrito con otro lenguaje de definición de datos, el *DDL conceptual.* Si se va a lograr la independencia física de los datos, entonces las definiciones conceptuales de DDL no deben comprender en lo absoluto ninguna consideración de la representación física ni de la técnica de acceso; deben ser *únicamente* definiciones del contenido de la información. Por lo tanto, en el esquema conceptual no debe haber ninguna referencia para la representación de campos almacenados, la secuencia de registros almacenados, los índices, los esquemas de dispersión, los apuntadores o cualquier otro detalle de almacenamiento y acceso. Si el esquema conceptual se hace verdaderamente independiente de los datos, entonces los esquemas externos, que están definidos en términos del esquema conceptual (vea la sección 2.6), también serán *forzosamente* independientes de los datos.

Entonces, la vista conceptual es una vista del contenido total de la base de datos, y el esquema conceptual es una definición de esa vista. Sin embargo, sería engañoso dar por hecho que el esquema conceptual no es nada más que un conjunto de definiciones muy similar a las definiciones que se encuentran (por ejemplo) en un programa COBOL actual. Las definiciones del esquema conceptual pretenden incluir muchas características adicionales, como las restricciones de seguridad y de integridad mencionadas en el capítulo 1. Algunas autoridades van más lejos al sugerir que el objetivo final del esquema conceptual es describir toda la empresa; no sólo los datos como tales, sino también la forma en que son utilizados, la forma en que fluyen de un punto a otro dentro de la empresa, su función en cada punto, lo controles de auditoría u otros que se aplican en cada punto, etcétera [2.3]. Sin embargo, debemos enfatizar que en la actualidad ningún sistema soporta realmente un esquema conceptual de cualquier cosa que se aproxime a este grado de amplitud; en la mayoría de los sistemas existentes, el "esquema conceptual" es en realidad algo más que una simple unión de todos los esquemas externos individuales más ciertas restricciones de seguridad y de integridad. Aunque en verdad es posible que los sistemas futuros sean mucho más sofisticados en cuanto al soporte del nivel conceptual.

2.5 EL NIVEL INTERNO

El tercer nivel de la arquitectura es el nivel interno. La **vista interna** es una representación de bajo nivel de toda la base de datos y consiste en muchas ocurrencias de cada uno de los diversos tipos de **registros internos.** "Registro interno" es el término de ANSI/SPARC para la construcción que hemos venido llamando registro *almacenado* (y que seguiremos utilizando). Por lo tanto, la vista interna está todavía distante del nivel físico, ya que no tiene que ver con términos como registros *físicos* —también denominados **bloques** o **páginas**— ni con ninguna consideración específica de los dispositivos, como el tamaño de los cilindros o de las pistas. En otras palabras, la vista interna en efecto da por hecho un espacio de direcciones lineal infinito; los detalles de cómo el espacio de direcciones se asocia con el almacenamiento físico, son en gran medida específicos del sistema y se omiten deliberadamente de la arquitectura general. *Nota:* El bloque. o página, es la **unidad de** E/S; es decir, es la cantidad de datos transferidos entre el almacenamiento secundario y la memoria principal en una sola operación de E/S. Los tamaños típicos de página son 1 KB, 2 KB o 4 KB (1 KB = 1024 bytes).

La vista interna se describe por medio del **esquema interno,** el cual no sólo define los diversos tipos de registros almacenados sino que especifica también qué índices existen, cómo están representados los campos almacenados, en qué secuencia están dichos registros, etcétera. (Una vez más, observe un ejemplo sencillo en la figura 2.2.) El esquema interno está escrito utilizando otro lenguaje más de definición de datos: el *DDL interno*. *Nota:* En este libro usaremos normalmente los términos más intuitivos "estructura de almacenamiento" o "base de datos almacenada" en lugar de "vista interna", así como el término "definición de la estructura de almacenamiento" en lugar de "esquema interno".

Para terminar, señalamos que, en ciertas situaciones excepcionales, a los programas de aplicación —en particular, las aplicaciones *de utilería* (vea la sección 2.11)— se les podría permitir operar directamente en el nivel interno en vez del nivel externo. Sobra decir que no es recomendable esta práctica, pues representa un riesgo para la seguridad (ya que se ignoran las restricciones de seguridad) y un riesgo para la integridad (debido a que, de igual manera, se ignoran las restricciones de integridad). Además, para iniciar, el programa será dependiente de los datos; aunque, en ocasiones, ésta podría ser la única forma de obtener la funcionalidad o el rendimiento requeridos (tal como le sucede al usuario de un lenguaje de programación de alto nivel que ocasionalmente tendría que descender al lenguaje ensamblador para satisfacer ciertos objetivos de funcionalidad o rendimiento).

2.6 TRANSFORMACIONES

Además de los tres niveles *corno tales,* la arquitectura de la figura 2.3 comprende ciertas **transformaciones;** en general, una transformación conceptual/interna y varias transformaciones externas/conceptual:

- La transformación *conceptual/interna* define la correspondencia entre la vista conceptual y la base de datos almacenada, y especifica cómo están representados los registros y campos conceptuales en el nivel interno. Si se modifica la estructura de la base de datos —es decir, si se hace un cambio a la definición de la estructura de almacenamiento—, entonces por consiguiente será necesario modificar la transformación conceptual/interna, de manera que

el esquema conceptual pueda permanecer invariable. (Por supuesto, es responsabilidad del DBA administrar dichos cambios.) En otras palabras, los efectos de dichos cambios deben aislarse por debajo del nivel conceptual, a fin de preservar la independencia física de los datos.

- La transformación *externa/conceptual* define la correspondencia entre una vista externa en particular y la vista conceptual. En general, las diferencias que puedan existir entre estos dos niveles son análogas a aquellas que puedan existir entre la vista conceptual y la base de datos almacenada. Por ejemplo, los campos pueden tener diferentes tipos de datos; los nombres de los registros y campos pueden ser cambiados; varios campos conceptuales pueden combinarse en un solo registro externo (virtual); etcétera. Puede existir cualquier cantidad de vistas externas al mismo tiempo; cualquier número de usuarios puede compartir una vista externa dada; es posible traslapar diferentes vistas externas.

Nota: Debe quedar claro que así como la transformación conceptual/interna es la clave para la independencia física de los datos, también las transformaciones externas/conceptual son la clave para la independencia **lógica** de los datos. Como vimos en el capítulo 1, un sistema proporciona la independencia física de los datos [1.3] si los usuarios y los programas de usuario son inmunes a los cambios en la estructura física de la base de datos almacenada. De igual manera, un sistema proporciona la independencia *lógica* de los datos [1.4] si los usuarios y los programas de usuario también son inmunes a los cambios en la estructura *lógica* de la base de datos (lo que significa cambios al nivel conceptual o "lógico de la comunidad"). En los capítulos 3 y 9, tendremos más que decir respecto de este tema importante.

Por cierto, la mayoría de los sistemas permiten que la definición de ciertas vistas externas se exprese en términos de otras (en efecto, mediante transformaciones *externas/externas),* en vez de requerir siempre una definición explícita de la transformación al nivel conceptual; una característica útil si varias vistas externas son parecidas entre sí. En particular, los sistemas relacionales brindan dicha posibilidad.

2.7 EL ADMINISTRADOR DE BASE DE DATOS

Como expliqué en el capítulo 1, el DA (administrador de *datos)* es la persona que toma las decisiones de estrategia y política con respecto a los datos de la empresa y el DBA (administrador de *base* de datos) es la persona que proporciona el apoyo técnico necesario para implementar dichas decisiones. Por lo tanto, el DBA es el responsable del control general del sistema al nivel técnico. Ahora podemos describir con un poco más de detalle algunas de las tareas del DBA. En general, estas tareas comprenden al menos todas las siguientes:

- *Definir el esquema conceptual*

  Es trabajo del administrador de datos decidir exactamente qué información contendrá la base de datos; en otras palabras, identificar las entidades de interés para la empresa e identificar la información que hay que registrar acerca de dichas entidades. Por lo regular a este proceso se le conoce como **diseño lógico** —en ocasiones *conceptual*— **de la base de datos.** Una vez que el administrador decidió el contenido de la base de datos a un nivel abstracto, entonces el DBA creará el esquema conceptual correspondiente, utilizando el DLL conceptual. El DBMS usará la forma objeto (compilada) de ese esquema para responder a las

peticiones de acceso. La forma fuente (sin compilar) actuará como documento de referencia para los usuarios del sistema.

Nota: En la práctica, las cosas pueden no ser tan claras como sugieren los señalamientos anteriores. En algunos casos, el administrador de datos podría crear directamente el esquema conceptual. En otras, el DBA podría hacer el diseño lógico.

■ *Definir el esquema interno*

El DBA también debe decidir la forma en que van a ser representados los datos en la base de datos almacenada. A este proceso se le conoce comúnmente como diseño físico de la base de datos.* Una vez realizado el diseño físico, el DBA deberá crear la definición de la estructura de almacenamiento correspondiente (es decir, el esquema interno), utilizando el DDL interno. Además, también deberá definir la transformación conceptual/interna asociada. En la práctica, es factible que uno de los dos DDLs (el conceptual o el interno; pero más probablemente el primero) incluya los medios para definir esa transformación; aunque las dos funciones (crear el esquema y definir la transformación) deben ser claramente separables. Al igual que el esquema conceptual, tanto el esquema interno como la transformación correspondiente existirán en las formas fuente y objeto.

■ *Establecer un enlace con los usuarios*

Es asunto del DBA enlazarse con los usuarios para asegurar que los datos necesarios estén disponibles y para escribir (o ayudar a escribir) los esquemas externos necesarios, utilizando el DDL externo aplicable. (Como ya mencionamos, un sistema dado podría manejar varios DDLs externos distintos.) Además, también es necesario definir las transformaciones externas/conceptual correspondientes. En la práctica, es probable que el DDL externo incluya los medios para especificar dichas transformaciones, pero, una vez más, los esquemas y las transformaciones deben ser claramente separables. Cada esquema externo, con la transformación correspondiente, existirá en las formas tanto fuente como objeto.

Otros aspectos de la función de enlace con los usuarios incluyen la asesoría sobre el diseño de aplicaciones; una capacitación técnica; ayuda en la determinación y resolución de problemas; así como otros servicios profesionales similares.

Definir las restricciones de seguridad y de integridad

Como ya expliqué, las restricciones de seguridad y de integridad pueden ser vistas como parte del esquema conceptual. El DDL conceptual debe incluir facilidades para especificar dichas restricciones.

■ *Definir las políticas de vaciado y recarga*

Una vez que una empresa se compromete con un sistema de base de datos, se vuelve drásticamente dependiente del funcionamiento exitoso de dicho sistema. En el caso de que se produzca un daño en cualquier parte de la base de datos —ocasionado, por ejemplo, por un error humano o por una falla en el hardware o en el sistema operativo— resulta esencial poder reparar los datos afectados con el mínimo de demora y con tan poco efecto como sea posible sobre el resto del sistema. Por ejemplo, de manera ideal no debería afectarse la disponibilidad de los datos que *no* fueron afectados. El DBA debe definir e implementar un esquema apropiado de control de daños que comprenda (a) la descarga o "vaciado" periódico

*Observe la secuencia: Primero decida qué datos desea, luego decida cómo representarlos en el almacenamiento. El diseño físico sólo deberá hacerse *después* de realizar el diseño lógico.

de la base de datos en un dispositivo de almacenamiento de respaldo y (b) la recarga de la base de datos cuando sea necesario, a partir del vaciado más reciente.

Por cierto, la necesidad de una rápida reparación de los datos es una de las razones por las que podría ser buena idea repartir todos los datos en varias bases de datos, en vez de mantenerlos todos en un mismo lugar; una base de datos individual podría muy bien definir una unidad para fines de vaciado y de recarga. En este sentido, observe que ya existen los *sistemas de terabytes** —es decir, los sistemas comerciales que almacenan más de un billón de bytes, aproximadamente— y se predice que los sistemas futuros serán mucho más grandes. Sobra decir que tales sistemas *VLDB* ("base de datos muy grande") requieren de una administración muy cuidadosa y sofisticada, en especial si hay necesidad de una disponibilidad continua (lo que sucede normalmente). Sin embargo, para efectos de simplicidad, seguiremos hablando como si de hecho sólo existiera una base de datos individual.

*■ *Supervisar el rendimiento y responder a los requerimientos cambiantes*

Como indiqué en el capítulo 1, el DBA es el responsable de organizar el sistema de tal manera que se obtenga el rendimiento "ideal para la empresa" y de hacer los ajustes apropiados —es decir, **afinar**— conforme las necesidades cambien. Por ejemplo, podría ser necesario **reorganizar** de vez en cuando la base de datos almacenada para asegurar que los niveles de rendimiento se mantengan aceptables. Como ya mencioné, todo cambio al nivel (interno) de almacenamiento físico debe estar acompañado por el cambio correspondiente en la definición de la transformación conceptual/interna, de manera que el esquema conceptual permanezca constante.

Desde luego, la anterior no es una lista detallada; simplemente pretende dar una idea del alcance y naturaleza de las responsabilidades del DBA.

2.8 EL SISTEMA DE ADMINISTRACIÓN / DE BASE DE DATOS

El DBMS **(sistema de administración de base de datos)** es el software que maneja todo acceso a la base de datos. De manera conceptual, lo que sucede es lo siguiente (consulte una vez más la figura 2.3):

1. Un usuario emite una petición de acceso, utilizando algún sublenguaje de datos específico (por lo regular SQL).
2. El DBMS intercepta esa petición y la analiza.
3. El DBMS inspecciona, en su momento, (las versiones objeto de) el esquema externo para ese usuario, la transformación externa/conceptual correspondiente, el esquema conceptual, la transformación conceptual/interna y la definición de la estructura de almacenamiento.
4. El DBMS ejecuta las operaciones necesarias sobre la base de datos almacenada.

---

*1024 bytes = 1 KB (kilobyte); 1024 KB = 1 MB (megabyte); 1024 MB = 1 GB (gigabyte); 1024 GB = 1 TB (terabyte); 1024 TB = 1 PB (petabyte); 1024 PB = 1 EB (exabyte). Tome nota que un gigabyte equivale aproximadamente a mil millones de bytes y que algunos textos en inglés emplean en ocasiones la abreviatura BB en lugar de GB.

A manera de ejemplo, considere lo que implica la recuperación de una ocurrencia de un registro externo en particular. En general, los campos serán solicitados desde varias ocurrencias de registros conceptuales, y cada ocurrencia de un registro conceptual solicitará a su vez campos de varias ocurrencias de registros almacenados. Entonces, de manera conceptual, el DBMS debe primero recuperar todas las ocurrencias solicitadas de los registros almacenados, luego construir las ocurrencias solicitadas de los registros conceptuales y después construir las ocurrencias solicitadas de los registros externos. En cada etapa, podrían ser necesarias conversiones de tipos de datos u otras.

Desde luego, la descripción anterior está muy simplificada; en particular, implica que todo el proceso es interpretativo, ya que asume que todo el proceso de analizar la petición, inspeccionar los diversos esquemas, etcétera, se realiza en tiempo de ejecución. La interpretación, por su parte, a menudo implica un rendimiento deficiente debido a la sobrecarga del tiempo de ejecución. Sin embargo, en la práctica podría ser posible *compilar* las peticiones de acceso antes del tiempo de ejecución (en particular, varios productos SQL actuales hacen esto; vea la anotación a las referencias [4.12] y [4.26] en el capítulo 4).

Analicemos ahora las funciones del DBMS con un poco más de detalle. Dichas funciones comprenderán por lo menos el manejo de todas las siguientes (consulte la figura 2.4):

- *Definición de datos*

  El DBMS debe ser capaz de aceptar definiciones de datos (esquemas externos, el esquema conceptual, el esquema interno y todas las transformaciones respectivas) en la forma fuente y convertirlas a la forma objeto correspondiente. En otras palabras, el DBMS debe incluir entre sus componentes un **procesador DDL,** o **compilador DDL,** para cada uno de los diversos DDLs (lenguajes de definición de datos). El DBMS también debe "entender" 1: definiciones DDL, en el sentido que, por ejemplo, "entienda" que los registros externos EMPLEADO incluyen un campo SALARIO; entonces, debe poder utilizar este conocimiento para analizar y responder a las peticiones de manipulación de datos (por ejemplo. "Obtener todos los empleados con salario < $50,000").

- *Manipulación de datos*

  El DBMS debe ser capaz de manejar peticiones para recuperar, actualizar o eliminar datos existentes en la base de datos o agregar nuevos datos a ésta. En otras palabras, el DBMS debe incluir un componente **procesador DML** o **compilador DML** para tratar con el DML (lenguaje de manipulación de datos).

- En general, las peticiones DML pueden ser "planeadas" o "no planeadas":

  a. Una petición **planeada** es aquella cuya necesidad fue prevista antes del momento de eje cutar la petición. Probablemente el DBA habrá afinado el diseño físico de la base de datos de tal forma que garantice un buen desempeño para las peticiones planeadas.

  b. En contraste, una petición **no planeada** es una consulta *ad hoc;* es decir, una petición para la que no se previó por adelantado su necesidad, sino que en vez de ello, surgió sin pensarlo. El diseño físico de la base de datos podría o no ser el adecuado para la peti ción específica en consideración.

  Para utilizar la terminología presentada en el capítulo 1 (sección 1.3), las peticiones planeadas son características de las aplicaciones "operacionales" o de "producción", **mientra** que las peticiones no planeadas son características de las aplicaciones de "apoyo a la **toma** de decisiones". Además, peticiones planeadas serán emitidas generalmente desde programas de aplicación preescritos, mientras que las no planeadas, por definición, serán emitidas

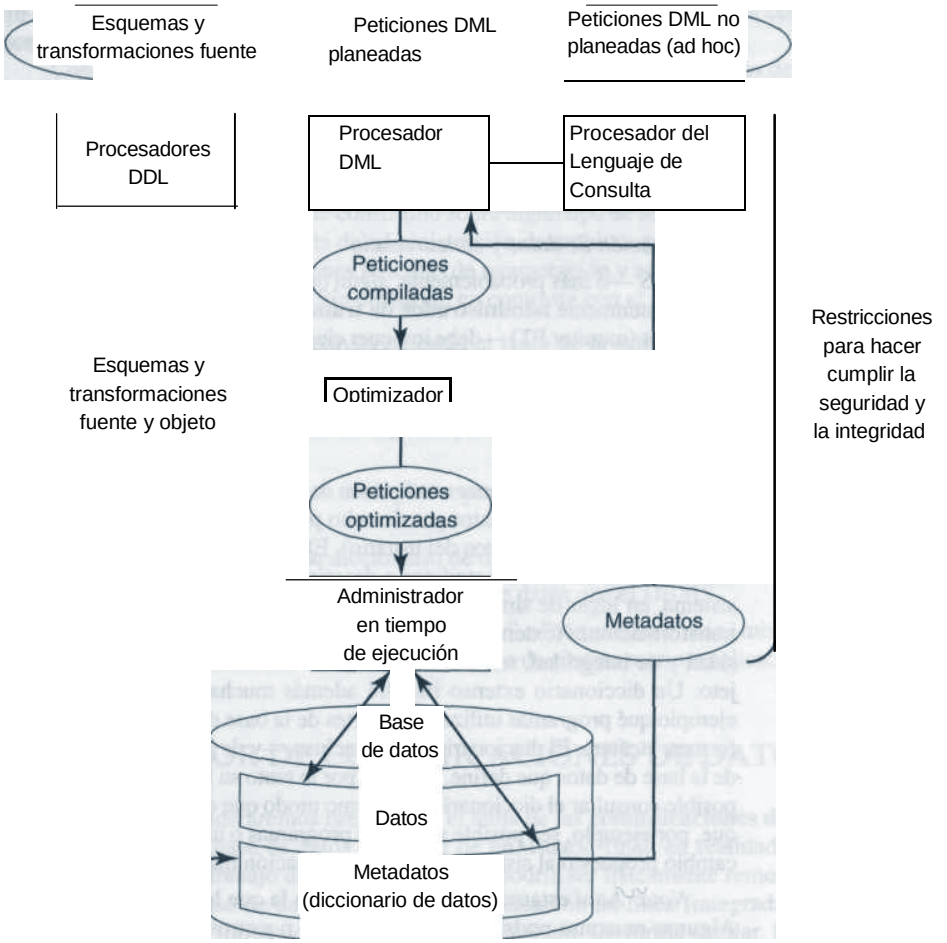


Figura 2.4 Funciones y componentes principales del DBMS.

en forma interactiva mediante algún **procesador del lenguaje de consulta.** *Nota:* Como vimos en el capítulo 1, el procesador del lenguaje de consulta es en realidad una aplicación integrada en línea, no una parte del DBMS *como tal;* lo incluimos en la figura 2.4 para ampliar la explicación.

Optimización y ejecución

Las peticiones DML, planeadas o no planeadas, deben ser procesadas por el componente **optimizador,** cuya finalidad es determinar una forma eficiente de implementar la petición. En el capítulo 17 explicamos la optimización en detalle. Las peticiones optimizadas se ejecutan entonces bajo el control del **administrador en tiempo de ejecución.** *Nota:* En la práctica,

el administrador en tiempo de ejecución recurrirá probablemente a algún tipo de *administrador de archivos* para acceder a los datos almacenados. Al final de esta sección explicamos brevemente los administradores de archivos.

- *Seguridad e integridad de los datos*

  El DBMS debe vigilar las peticiones del usuario y rechazar todo intento de violar las restricciones de seguridad y de integridad definidas por el DBA (vea la sección anterior). Estas tareas pueden realizarse durante el tiempo de compilación, de ejecución o entre ambos.

- *Recuperación de datos y concurrencia*

  El DBMS —o más probablemente, algún otro componente de software relacionado, denominado comúnmente **administrador de transacciones** o **monitor de procesamiento de transacciones** (monitor PT)— debe imponer ciertos controles de recuperación y concurrencia. Los detalles de estos aspectos del sistema están fuera del alcance de este capítulo; para una explicación más a fondo, consulte la parte IV de este libro. *Nota:* El administrador de transacciones no se muestra en la figura 2.4 debido a que, por lo regular, no forma parte del DBMS *como tal.*

- *Diccionario de datos*

  El DBMS debe proporcionar una función de **diccionario de datos.** Este diccionario puede ser visto como una base de datos por derecho propio (aunque una base de datos del sistema mí que como una base de datos del usuario). El diccionario contiene "datos acerca de los datos" (en ocasiones llamados *metadatos* o *descriptores);* es decir, *definiciones* de otros objetos del sistema, en lugar de simples "datos en bruto". En particular, todos los diversos esquemas y transformaciones (externos, conceptuales, etcétera) y todas las diversas restricciones de seguridad y de integridad, serán almacenadas en el diccionario, tanto en forma fuente como objeto. Un diccionario extenso incluirá además mucha información adicional; mostrará por ejemplo qué programas utilizan qué partes de la base de datos, qué usuarios necesitan qué informes, etcétera. El diccionario podría incluso —y de hecho, debería— estar integrado dentro de la base de datos que define, e incluir por lo tanto su propia definición. En realidad, debe ser posible consultar el diccionario del mismo modo que cualquier otra base de datos, de maner que, por ejemplo, sea posible saber qué programas o usuarios se podrían ver afectados por un cambio propuesto al sistema. Para una explicación más amplia, consulte el capítulo 3.

  Nota: Aquí estamos tocando un área en la que hay mucha confusión de terminología. Algunas personas podrían referirse a lo que nosotros llamamos diccionario como ***directorio*** o *catálogo* —con la implicación de que los directorios y catálogos son, en cierta forma. inferiores a un verdadero diccionario— y podrían reservar el término *diccionario* para hacer referencia a una clase específica (importante) de herramienta de desarrollo de aplicaciones. Otros términos que también son utilizados, a veces, para hacer referencia a este último tipo de objeto son *depósito de datos* (vea el capítulo 13) y *enciclopedia de datos.*

- *Rendimiento*

  Sobra decir que el DBMS debe realizar todas las tareas antes identificadas de la manera más eficiente posible.

Podemos resumir todo lo anterior diciendo que la finalidad general del DBMS consiste en proporcionar una **interfaz de usuario** para el sistema de base de datos. Podemos definir la interfaz de usuario como un límite en el sistema debajo del cual todo es invisible para el usuario. Por lo tanto, por definición la interfaz de usuario se encuentra en el nivel *externo.* Sin embargo, como veremos en el capítulo 9, existen algunas situaciones en las que es poco probable que la

vista externa difiera de manera significativa de la parte relevante de la vista conceptual subyacente, por lo menos en los productos comerciales SQL actuales.

Concluimos esta sección con una breve comparación entre los sistemas de administración de bases de datos y los sistemas de administración de **archivos** *(administradores de archivos,* o *servidores de archivos,* para abreviar). En esencia, el administrador de archivos es el componente del sistema operativo subyacente que administra los archivos almacenados; por lo tanto, hablando en términos generales, está "más cerca del disco" de lo que lo está el DBMS. (De hecho, el DBMS es generalmente construido sobre algún tipo de administrador de archivos.) Por lo tanto, el usuario de un sistema de administración de archivos podrá crear y destruir archivos almacenados y realizar operaciones sencillas de recuperación y actualización sobre registros almacenados en dichos archivos. Sin embargo, en contraste con el DBMS:

- Los administradores de archivos no están al tanto de la estructura interna de los registros almacenados, de ahí que no puedan manejar peticiones que se basen en el conocimiento de esa estructura.
- Por lo regular ofrecen poco o ningún soporte para las restricciones de seguridad y de in tegridad.
- Por lo regular ofrecen poco o ningún soporte para los controles de recuperación y con currencia.
- No hay un concepto real de diccionario de datos en el nivel del administrador de archivos.
- Proporcionan mucho menos independencia de datos que el DBMS.
- Por lo regular los archivos no están "integrados" o "compartidos" en el mismo sentido que en una base de datos (normalmente son exclusivos de cierto usuario o aplicación en particular).

2.9 EL ADMINISTRADOR DE COMUNICACIONES DE DATOS

En esta sección consideraremos brevemente el tema de las **comunicaciones de datos.** Las peticiones emitidas a la base de datos por parte de un usuario final, en realidad son transmitidas desde la estación de trabajo del usuario —la cual podría ser físicamente remota con respecto al propio sistema de base de datos— hacia cierta aplicación en línea (integrada u otra) y de ahí hacia el DBMS en la forma de *mensajes de comunicación.* De forma similar, las respuestas que regresan del DBMS y la aplicación en línea hacia la estación de trabajo del usuario, también son transmitidas en forma de dichos mensajes. Todas estas transmisiones de mensajes se llevan a cabo bajo el control de otro componente de software, el **administrador de comunicaciones de datos** (administrador CD).

Este administrador no forma parte del DBMS, sino que es un sistema autónomo por derecho propio. Sin embargo, puesto que es claramente necesario que trabaje en armonía con el DBMS, en ocasiones se les considera como socios igualitarios en una empresa de más alto nivel denominada **sistema de base de datos/comunicaciones de datos** (sistema BD/CD); en el cual el DBMS se ocupa de la base de datos y el administrador CD maneja todos los mensajes hacia y desde el DBMS, o más precisamente, hacia y desde las aplicaciones que el DBMS utiliza. No obstante, en este libro tendremos relativamente poco que decir con respecto al manejo de mensajes como tal (por sí solo, es un tema muy extenso). En la sección 2.12 explicamos brevemente la cuestión de la comunicación *entre sistemas distintos* (es decir, entre máquinas distintas en una red de comunicaciones, como Internet); aunque en realidad ése es un tema aparte.

2.10 ARQUITECTURA CLIENTE-SERVIDOR

Hasta ahora, en este capítulo hemos venido explicando los sistemas de bases de datos desde el punto de vista de la así llamada arquitectura ANSI/SPARC. En la figura 2.3 en particular, presentamos una imagen simplificada de esta arquitectura. En esta sección estudiaremos los sistemas de bases de datos desde una perspectiva ligeramente diferente. Desde luego, la finalidad principal de dichos sistemas es apoyar el desarrollo y la ejecución de las aplicaciones de bases de datos. Por lo tanto, desde un punto de vista más elevado, un sistema de base de datos puede ser visto como un sistema que tiene una estructura muy sencilla de dos partes, las cuales consisten en un **servidor** (también denominado *parte dorsal o servicios de fondo)* y un conjunto de **clientes** (también llamados *partes frontales, aplicaciones para el usuario o interfaces).* Consulte la figura 2.5. *Explicación:*

- El *servidor* es precisamente el propio DBMS. Soporta todas las funciones básicas del DBMS expuestas en la sección 2.8: definición de datos, manipulación de datos, seguridad e integri dad de los datos, etcétera. En particular, proporciona todo el soporte de los niveles externo, conceptual e interno explicados en las secciones 2.3 a 2.6. Por lo tanto, en este contexto, "servidor" es sólo el nombre del DBMS.
- Los *clientes* son las diversas aplicaciones que se ejecutan sobre el DBMS, tanto aplicaciones escritas por el usuario como aplicaciones integradas (es decir, aplicaciones proporcionadas por el fabricante del DBMS o por alguna otra compañía). Por supuesto, en lo que concierne al servidor, no hay diferencia entre las aplicaciones escritas por el usuario y las integradas; todas usan la misma interfaz con el servidor, como la interfaz de nivel externo expuesta en la sección 2.3.

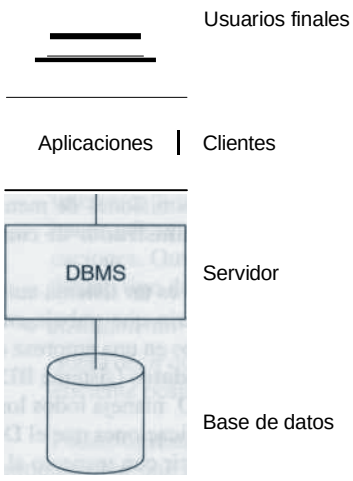


Figura 2.5 Arquitectura cliente-servidor.

Nota: Ciertas aplicaciones especiales de "utilería" podrían constituir una excepción a lo anterior, ya que en ocasiones podrían necesitar operar directamente en el nivel *interno* del sistema (como mencioné en la sección 2.5). Estas utilerías son consideradas más bien como componentes integrales del DBMS, en vez de aplicaciones en el sentido usual. Las explico con más detalle en la siguiente sección.

Profundizaremos un poco sobre la cuestión de aplicaciones escritas por el usuario en comparación con las aplicaciones proporcionadas por el fabricante:

- Las *aplicaciones escritas por el usuario* son en esencia programas de aplicación comunes, escritos por lo regular ya sea en un lenguaje convencional de tercera generación (como C o COBOL) o en algún lenguaje de tipo propietario de cuarta generación; aunque en ambos casos es necesario acoplar de alguna manera el lenguaje con un sublenguaje de datos apropiado, como expliqué en la sección 2.3.
- Las *aplicaciones proporcionadas por el fabricante* (a menudo llamadas **herramientas)** son aplicaciones cuya finalidad básica es auxiliar en la creación y ejecución de otras aplicaciones. Las aplicaciones creadas son aplicaciones confeccionadas para alguna tarea específica (podrían no parecer aplicaciones en el sentido convencional; de hecho, la idea general de las herramientas es permitir a los usuarios, en especial a los usuarios finales, la creación de aplicaciones *sin* tener que escribir programas en un lenguaje de programación convencional). Por ejemplo, una herramienta proporcionada por el fabricante sería un *generador de informes,* cuyo propósito es permitir a los usuarios obtener del sistema informes con cierto formato. Toda petición de informe puede verse como un pequeño programa de aplicación, escrito en un *lenguaje generador de informes* de muy alto nivel (y de finalidad especial).

  Las herramientas proporcionadas por el fabricante pueden dividirse en varias clases relativamente distintas:

  a. Procesadores de lenguaje de consulta;
  b. Generadores de informes;
  c. Subsistemas de gráficos comerciales;
  d. Hojas de cálculo;
  e. Procesadores de lenguaje natural;
  f. Paquetes estadísticos;
  g. Herramientas de administración de copias o "extracción de datos";
  h. Generadores de aplicaciones (incluyen procesadores 4GL);
  i. Otras herramientas de desarrollo de aplicaciones, incluyendo productos de ingeniería de software asistida por computadora (CASE);

  y muchos otros. Los detalles de dichas herramientas exceden el alcance de este libro; sin embargo, señalaremos que (como dijimos antes) debido a que la idea general de un sistema de base de datos es apoyar la creación y ejecución de aplicaciones, la calidad de las herramientas disponibles es, o debería ser, un factor primordial en "la decisión de la base de datos" (es decir, en el proceso de seleccionar un nuevo producto de base de datos). En otras palabras, el DBMS *como tal* no es el único factor a tomar en consideración, ni tampoco es necesariamente el factor más importante.

Cerramos esta sección con una referencia anticipada. Puesto que el sistema en su conjunto puede ser dividido claramente en dos partes (clientes y servidores), surge la posibilidad de

operar los dos en *máquinas diferentes.* En otras palabras, existe el potencial para el **procesamiento distribuido.** Procesamiento distribuido significa que es posible conectar distintas máquinas en cierto tipo de red de comunicaciones, de tal manera que una sola tarea de procesamiento de datos pueda dividirse entre varias máquinas de la red. De hecho, esta posibilidad es tan atractiva —por diversas razones, principalmente económicas— que el término "cliente-servidor" ha llegado a aplicarse casi exclusivamente al caso en el que el cliente y el servidor están, en efecto, en máquinas distintas. En la sección 2.12 explicaré con más detalle el procesamiento distribuido.

2.11 UTILERÍAS

Las utilerías son programas diseñados para ayudar al DBA en sus numerosas tareas administrativas. Como mencioné en la sección anterior, algunos programas de utilería operan en el nivel externo del sistema y en realidad no son más que aplicaciones de propósito especial; algunas podrían incluso no ser proporcionadas por el fabricante del DBMS, sino por alguna otra compañía. Sin embargo, otras utilerías operan directamente en el nivel interno (en otras palabras, son realmente parte del servidor), de ahí que deban ser proporcionadas al fabricante del DBMS.

Aquí hay algunos ejemplos del tipo de utilerías que comúnmente son requeridas en la práctica:

- Rutinas de **carga,** para crear la versión inicial de la base de datos a partir de uno o más archivos del sistema operativo;
- Rutinas de **descarga/recarga,** para descargar la base de datos (o partes de ella), para respaldar los datos almacenados y para recargar datos desde dichas copias de respaldo (por **supuesto,** la "rutina de recarga" es básicamente idéntica a la rutina de carga recién mencionada);
- Rutinas de **reorganización,** para reordenar los datos en la base de datos almacenada, por distintas razones que normalmente tienen que ver con el desempeño; por ejemplo, agrupar datos en el disco de alguna forma en particular o recuperar espacio ocupado por datos que se volvieron obsoletos;
- Rutinas **estadísticas,** para calcular diversas estadísticas de desempeño, como el tamaño de los archivos, las distribuciones de valores, los contadores de operaciones de E/S, etcétera:
- Rutinas de **análisis,** para analizar las estadísticas arriba mencionadas;

y así sucesivamente. Por supuesto, esta lista representa sólo una pequeña muestra del rango de funciones que ofrecen generalmente las utilerías; existe una gran cantidad de otras posibilidades.

2.12 EL PROCESAMIENTO DISTRIBUIDO

Para repetir lo dicho en la sección 2.10, el término "procesamiento distribuido" significa que distintas máquinas pueden conectarse en una red de comunicaciones como Internet, de tal manera que una sola tarea de procesamiento de datos pueda extenderse a varias máquinas de la red. (En ocasiones, también se usa el término "procesamiento paralelo" básicamente con el mismo significado, con excepción de que las máquinas distintas tienden a estar físicamente muy cerca en un sistema "paralelo", lo que no es necesario en un sistema "distribuido"; en último caso, por

ejemplo, podrían estar geográficamente dispersas). La comunicación entre las diversas máquinas es manejada mediante algún tipo de software de administración de redes; tal vez una extensión del administrador CD que explicamos en la sección 2.9 y más probablemente un componente de software independiente.

El procesamiento distribuido presenta muchos niveles o variedades posibles. Para repetir lo dicho en la sección 2.10, un caso sencillo comprendería la operación de los servicios dorsales del DBMS (el servidor) en una máquina y las aplicaciones de usuario (los clientes) en otra. Consulte la figura 2.6.

Como mencioné al final de la sección 2.10, aunque estrictamente hablando el término "cliente-servidor" es puramente arquitectónico. Se ha convertido casi en sinónimo de la organización ilustrada en la figura 2.6, en la que un cliente y un servidor se ejecutan en máquinas diferentes. De hecho, hay muchos argumentos a favor de dicho esquema:

- El primero es básicamente el simple argumento de procesamiento paralelo normal: es decir, ahora se están aplicando muchas unidades de procesamiento para las tareas en conjunto, mientras que el procesamiento del servidor (base de datos) y del cliente (aplicación) se están haciendo en paralelo. De ahí que el tiempo de respuesta y la velocidad real de transporte tengan mejorías.
- Además, la máquina servidor podría ser una máquina construida a la medida y adaptada a la función del DBMS (una "máquina de base de datos") y podría, por lo tanto, proporcionar un mejor desempeño del DBMS.

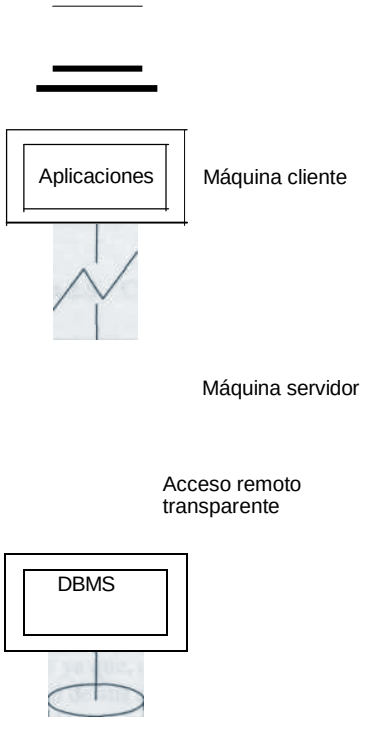


Figura 2.6 Cliente y servidor operando en máquinas diferentes.

- En forma similar, la máquina cliente podría ser una estación de trabajo personal adaptada a las necesidades del usuario final y por lo tanto, capaz de proporcionar mejores interfaces, una alta disponibilidad, respuestas más rápidas y en general una mejor facilidad de uso para el usuario.
- Varias máquinas cliente distintas podrían ser (de hecho serían) capaces de acceder a la misma máquina servidor. Por lo tanto, una sola base de datos podría ser compartida entre varios sistemas cliente distintos (vea la figura 2.7).

Además de los argumentos anteriores, está el punto de que ejecutar los clientes y el servidor en máquinas separadas coincide con la forma en que operan en realidad las empresas. Es muy común que una sola empresa —por ejemplo, un banco— opere muchas computadoras, de tal modo que los datos de una parte de la empresa estén almacenados en una computadora y los datos de otra parte estén almacenados en otra computadora. También es bastante común que los usuarios de una computadora necesiten acceso por lo menos ocasional a los datos almacenados en otra computadora. Siguiendo con el ejemplo del banco, es muy probable que los usuarios de una sucursal necesiten acceder ocasionalmente a los datos almacenados en otra. Observe, por lo tanto, que las máquinas cliente podrían tener almacenados datos propios y que la máquina servidor podría tener sus propias aplicaciones. Por lo tanto, es común que cada máquina actúe como servidor para ciertos usuarios y como cliente para otros (vea la figura 2.8); en otras palabras, cada máquina soportará un *sistema de base de datos completo,* en el sentido al que se refieren las secciones anteriores de este capítulo.

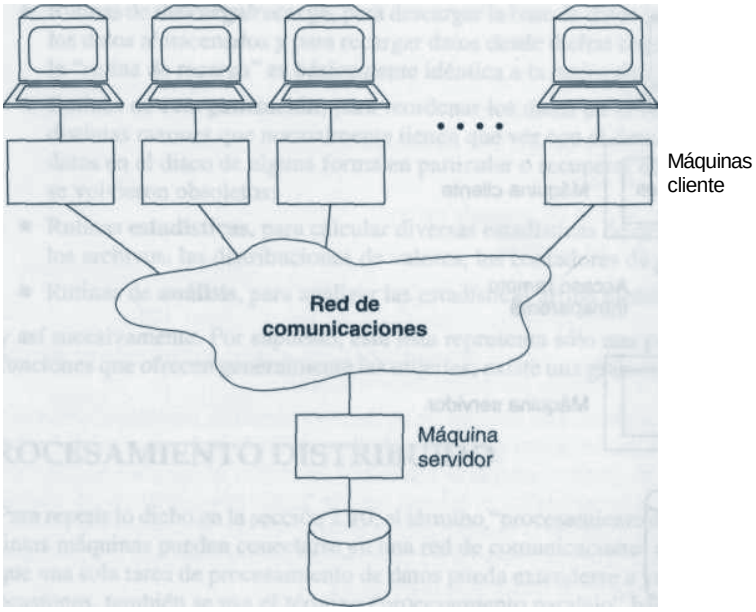


Figura 2.7 Una máquina servidor, varias máquinas cliente.

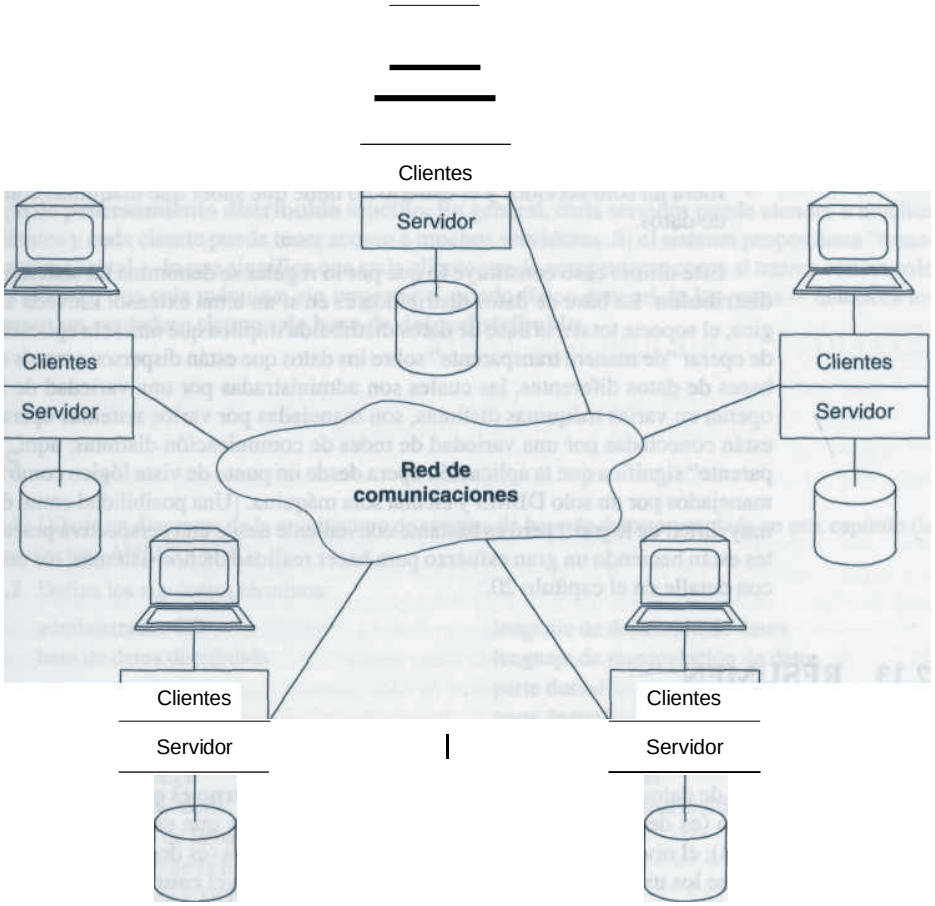


Figura 2.8 Cada máquina opera como clientes y como servidor.

La idea final es que una sola máquina cliente podría ser capaz de acceder a varias máquinas servidor diferentes (lo contrario al caso ilustrado en la figura 2.7). Esta posibilidad es conveniente ya que, como mencioné antes, las empresas operan por lo regular de tal manera que la totalidad de sus datos no están almacenados en una sola máquina, sino más bien están esparcidos a través de muchas máquinas distintas, y las aplicaciones necesitarán a veces tener la posibilidad de acceder a los datos de más de una máquina. Básicamente, este acceso puede ser proporcionado en dos formas:

Una máquina cliente dada podría ser capaz de acceder a cualquier cantidad de servidores, pero sólo uno a la vez (es decir, cada petición individual de base de datos debe ser dirigida

a un solo servidor). En un sistema así, no es posible combinar datos de dos o más servidores con una sola petición. Además, el usuario de dicho sistema debe saber qué máquina en particular contiene qué piezas de datos.

- El cliente podría ser capaz de acceder a varios servidores en forma simultánea (es decir, una sola petición de base de datos podría combinar datos de varios servidores). En este caso, los servidores ven al cliente —desde un punto de vista lógico— como si en realidad fuera un solo servidor y el usuario no tiene que saber qué máquinas contienen qué piezas de datos.

Este último caso constituye lo que por lo regular se denomina un **sistema de** base de datos **distribuida.** La base de datos distribuida es en sí un tema extenso. Llevada a su conclusión lógica, el soporte total a la base de datos distribuida implica que una sola aplicación debe ser capaz de operar "de manera transparente" sobre los datos que están dispersos a través de una variedad de bases de datos diferentes, las cuales son administradas por una variedad de DBMSs distintos. operan en varias máquinas distintas, son manejadas por varios sistemas operativos diferentes y están conectadas por una variedad de redes de comunicación distintas; aquí, "de manera transparente" significa que la aplicación opera desde un punto de vista lógico como si los datos fueran manejados por un solo DBMS y en una sola máquina. ¡Una posibilidad como ésta podría parecer muy difícil de lograr!; pero es bastante conveniente desde una perspectiva práctica, y los fabricantes están haciendo un gran esfuerzo para hacer realidad dichos sistemas, los cuales explicaremos con detalle en el capítulo 20.

2.13 RESUMEN

En este capítulo hemos visto los sistemas de bases de datos desde el punto de vista de la arquitectura. Primero describimos la **arquitectura ANSI/SPARC,** la cual divide a un sistema de base de datos en tres niveles, como sigue: El nivel **interno** es el más cercano al almacenamiento físico (es decir, es aquel que se ocupa de la forma en que están almacenados físicamente los datos); el nivel **externo** es el más cercano a los usuarios (es decir, es el que se ocupa de la forma en que los usuarios individuales ven los datos); y el nivel **conceptual** es un nivel de indirección entre los otros dos (proporciona una *vista comunitaria* de los datos). Los datos, como se perciben en cada nivel, están descritos por medio de un **esquema** (o por varios esquemas, en el caso del nivel externo). Las **transformaciones** definen la correspondencia entre (a) un esquema externo dado y el esquema conceptual y (b) el esquema conceptual y el esquema interno. Estas transformaciones son la clave para proporcionar la **independencia lógica** y **física de** los datos, respectivamente.

Los usuarios —es decir, los usuarios finales y los programadores de aplicaciones, los cuales operan al nivel externo— interactúan con los datos por medio de un **sublenguaje** de datos, el cual se divide por lo menos en dos componentes: un DDL **(lenguaje de definición de** datos) y un DML **(lenguaje de manipulación de datos).** El sublenguaje de datos está incrustado en un **lenguaje** anfitrión. *Nota:* Los límites entre el lenguaje anfitrión y el sublenguaje de datos, y entre el DDL y el DML, son principalmente de naturaleza conceptual; en forma ideal, deberían ser "transparentes para el usuario".

También vimos más de cerca las funciones del **DBA** y del **DBMS.** Entre otras cosas, el DBA es el responsable de crear el esquema interno (el **diseño físico de la base de datos);** en contraste,

la creación del esquema conceptual (el **diseño lógico** o **conceptual de la base de datos)** es responsabilidad del administrador de *datos.* Y el DBMS es responsable, entre otras cosas, de implementar las peticiones DDL y DML del usuario. El DBMS también es responsable de proporcionar cierto tipo de función de **diccionario de datos.**

Otra forma conveniente de ver a los sistemas de bases de datos es como si consistieran en un **servidor** (el propio DBMS) y un conjunto de **clientes** (las aplicaciones). Los clientes y servidores pueden operar, y a menudo lo harán, en máquinas independientes, proporcionando así un tipo de **procesamiento distribuido** sencillo. En general, cada servidor puede atender a muchos clientes y cada cliente puede tener acceso a muchos servidores. Si el sistema proporciona "transparencia" total —lo que significa que cada cliente puede comportarse como si tratara con un solo servidor en una sola máquina, sin importar el estado físico general de las cosas— entonces tenemos un verdadero **sistema de base de datos distribuida.**

EJERCICIOS

2.1 Dibuje un diagrama de la arquitectura de sistema de base de datos presentada en este capítulo (la arquitectura ANSI/SPARC).

2.2 Defina los siguientes términos:

administrador CD
base de datos distribuida
carga cliente
definición de la estructura de almacenamiento
descarga/recarga diccionario de datos diseño físico de la base de datos diseño lógico de la base de datos esquema, vista y DDL conceptuales esquema, vista y DDL externos esquema, vista y DDL internos interfaz de usuario
lenguaje anfitrión
lenguaje de definición de datos
lenguaje de manipulación de datos
parte dorsal parte frontal
petición no planeada
petición planeada
procesamiento distribuido
reorganización
servidor
sistema BD/CD
sublenguaje de datos
transformación conceptual/interna
transformación externa/conceptual
utilería

2.3 Explique la secuencia de pasos comprendidos en la recuperación de una ocurrencia de un registro externo específico.

2.4 Liste las principales funciones que realiza el DBMS.

2.5 Haga una distinción entre la independencia lógica y física de los datos.

2.6 ¿Qué entiende por el término *metadatos?*

2.7 Liste las principales funciones que realiza el DBA.

2.8 Haga una distinción entre el DBMS y un sistema de administración de archivos.

2.9 Dé algunos ejemplos de herramientas proporcionadas por el fabricante.

2.10 Dé algunos ejemplos de utilerías de base de datos.

2.11 Examine cualquier sistema de base de datos que pudiera tener disponible. Procure transformar ese sistema a la arquitectura ANSI/SPARC tal como se describe en este capítulo. ¿Maneja claramente los tres niveles de la arquitectura? ¿Cómo están definidas las transformaciones entre niveles? ¿Cómo lucen los diversos DDLs (externo, conceptual, interno)? ¿Qué sublenguajes de datos maneja el sistema? ¿Qué lenguajes host? ¿Quién realiza la función de DBA? ¿Existen algunas herramientas de seguri dad y de integridad? ¿Hay un diccionario de datos? ¿Se describe éste a sí mismo? ¿Qué aplicaciones proporcionadas por el fabricante soporta el sistema? ¿Qué utilerías? ¿Hay un administrador indepen diente de comunicaciones de datos ? ¿Existe alguna posibilidad de procesamiento distribuido?

REFERENCIAS Y BIBLIOGRAFÍA

Las referencias siguientes son a la fecha bastante antiguas (con excepción de la última), pero aún son importantes para los conceptos presentados en este capítulo.

2.1 ANSI/X3/SPARC Study Group on Data Base Management Systems: Interim Report, FD7(bulletin of ACM SIGMOD) 7, No. 2 (1975).

2.2 Dionysios C. Tsichritzis y Anthony Klug (eds.): "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems", *Information Systems 3* (1978).

Las referencias [2.1-2.2] corresponden a los informes provisional y final, respectivamente, del tan mencionado Grupo de Estudio ANSI/SPARC. El Grupo de Estudio sobre Sistemas de Administración de Base de Datos ANSI/X3/SPARC (para dar su nombre completo), fue establecido a finales de 1972 por el SPARC (Comité de Planeación de Estándares y Requerimientos) del Comité sobre Computadoras y Procesamiento de Información (X3) del ANSI (Instituto Nacional Americano de Estándares). (Alrededor de 25 años más tarde, el nombre X3 se cambió por NCITS: Comité Nacional sobre Estándares en Tecnología de la Información.) Los objetivos del Grupo de Estudio fueron determinar qué áreas, si las había, de la tecnología de base de datos eran las adecuadas para estandarizar y producir un conjunto de recomendaciones de acción en cada una de ellas. Al trabajar para alcanzar estos objetivos, el Grupo de Estudio adoptó la posición de que las *interfaces* eran el único aspecto de un sistema de base de datos que podría ser sujeto a la estandarización y de acuerdo con ello, definió una arquitectura generalizada de sistema de base de datos, o infraestructura, que enfatizaba el papel de dichas interfaces. El informe final ofrece una descripción detallada de esa arquitectura y de algunas de las 42 (!) interfaces identificadas. El informe provisional es un documento de trabajo anterior que sigue teniendo cierto interés; proporciona detalles adicionales en algunas áreas.

2.3 J. J. van Griethuysen (ed.): "Concepts and Terminology for the Conceptual Schema and the In formation Base", International Organization for Standardization (ISO) Technical Report ISO/TR 9007:1987(E) (marzo, 1982; revisado en julio, 1987).

Este documento es un informe de un grupo de trabajo de la ISO (Organización Internacional de Estándares) cuyos objetivos comprendían "la definición de conceptos para lenguajes de esquema conceptual". Incluye una introducción a los tres candidatos en competencia (para ser más precisos, a los tres *conjuntos* de candidatos) para un conjunto apropiado de formalismos, y aplica cada uno de los tres a un ejemplo común que comprende las actividades de una autoridad ficticia de registro de automóviles. Los tres conjuntos de competidores son (1) esquemas de "entidad-atribu-

to-vínculo", (2) esquemas de "vínculo binario" y (3) esquemas de "lógica de predicados interpretados". El informe también incluye una exposición de los conceptos fundamentales subyacentes a la noción del esquema conceptual y sugiere algunos principios como base para la implementatión de un sistema que soporte en forma adecuada dicha noción. Aunque en ocasiones resulta denso, es un documento importante para todo el que se interese seriamente en el nivel conceptual del sistema.

2.4 William Kent: *Data and Reality.* Amsterdam, Netherlands: North-Holland/NuevaYork, N.Y.: Elsevier Science (1978).

Una explicación estimulante e incitante a la reflexión sobre la naturaleza de la información y en particular, sobre el esquema conceptual. "Este libro proyecta una filosofía de que la vida y la realidad son en el fondo amorfas, desordenadas, contradictorias, inconsistentes, irracionales y no objetivas" (extracto del último capítulo). Se puede ver al libro en gran parte como un compendio de problemas reales con los que (se sugiere) los formalismos de las bases de datos existentes tienen dificultades para enfrentar; en particular, los formalismos que se basan en estructuras del tipo de registro convencional, que incluyen al modelo relational. Recomendado.

2.5 Odysseas G. Tsatalos, Marvin H. Solomon y Yannis E. Ioannidis: "The GMAP: A Versatile Tool for Physical Data Independence", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

Las siglas GMAP corresponden a *Ruta de Acceso Generalizada de Nivel Múltiple.* Los autores de este artículo señalan correctamente que los productos de base de datos actuales "obligan a los usuarios a encasillar sus consultas en términos de un esquema lógico que está ligado en forma directa a las estructuras físicas", de ahí que sean más bien débiles en cuanto a la independencia de los datos. Por lo tanto, en el artículo proponen un lenguaje de transformación conceptual/interna (para emplear la terminología del presente capítulo), que puede ser usada para especificar muchas más clases de transformaciones que las que se manejan generalmente en los productos actuales. Dado un "esquema lógico, en particular el lenguaje (que está basado en el álgebra relational —vea el capítulo 6— y por lo tanto es de naturaleza declarativa, no de procedimientos) permite la especificación de numerosos esquemas físicos diferentes, todos ellos derivados formalmente de un esquema lógico. Entre otras cosas, dichos esquemas físicos pueden incluir particiones verticales y horizontales (vea el capítulo 20), cualquier cantidad de rutas de acceso físico, agrupamiento y redundancia controlada.

El artículo también ofrece un algoritmo para transformar las operaciones del usuario, frente al esquema lógico, en operaciones equivalentes, frente al esquema físico. Una implementatión prototipo muestra que el DBA puede ajustar el esquema físico para lograr "un rendimiento significativamente mejor al que es posible obtener con las técnicas convencionales".

CAPITULO 3

Una introducción a las bases de datos relacionales

3.1 INTRODUCCIÓN

Como expliqué en el capítulo 1, en este libro hago énfasis en gran medida en las bases de datos relacionales. En particular, la parte II cubre los fundamentos teóricos de dichos sistemas (el modelo relacional) con una profundidad considerable. La finalidad del presente capítulo consiste simplemente en dar una introducción preliminar, intuitiva y muy informal para el material que abordaremos en la parte II (y hasta cierto punto, también en las secciones subsiguientes), con el fin de allanar el camino para una mejor comprensión de las secciones posteriores del libro. En los capítulos posteriores explicaremos la mayoría de los temas mencionados de manera más formal y con mayor detalle.

3.2 UNA MIRADA INFORMAL AL MODELO RELACIONAL

Hemos mencionado en varias ocasiones que los sistemas relacionales se basan en un fundamento formal, o teoría, denominado *el modelo relacional de datos.* De manera intuitiva, lo que esta afirmación significa (entre otras cosas) es que en dichos sistemas:

1. **Aspecto estructural:** El usuario percibe la información de la base de datos como tablas y nada más que tablas;
2. **Aspecto de integridad:** Estas tablas satisfacen ciertas restricciones de integridad (que explicaremos hacia el final de esta sección);
3. **Aspecto de manipulación:** Los operadores disponibles para que el usuario manipule estas tablas —por ejemplo, para fines de recuperación de datos— son operadores que derivan tablas a partir de tablas. En particular, tres de estos operadores son importantes: *restringir, proyectar y juntar* (este último operador también es conocido como combinar o reunir).

La figura 3.1 muestra una base de datos relacional muy sencilla, la base de datos de departamentos y empleados. Como puede ver, la base de datos es en efecto "percibida como tablas" (y los significados de dichas tablas pretenden ser evidentes pos sí mismos).

DEPTO

DEPTO#	NOMDEPTO	PRESUPUESTO
D1	Comercialización	**10M**
D2	Desarrollo	**12M**
D3	Investigación	5M

EMP

EMP#	NOMEMP	DEPTO#	SALARIO
E1	López Cheng	D1	**40K**
E2	Pérez	D1	**42K**
E3	Hernández	D2	30K
E4		D2	35K

Figura 3.1 La base de datos de departamentos y empleados (valores de ejemplo).

La figura 3.2 muestra algunos ejemplos de las operaciones restringir, proyectar y juntar aplicadas a la base de datos de la figura 3.1. Las siguientes son (de manera muy vaga) las definiciones de dichas operaciones:

- La operación **restringir** (también conocida como seleccionar) extrae las filas especificadas de una tabla.
- La operación **proyectar** extrae las columnas especificadas de una tabla.
- La operación **juntar** reúne dos tablas con base en valores comunes de una columna común.

Restringir:

DEPTOs donde PRESUPUESTO > 8M

Resultado:

DEPTO#	NOMDEPTO	PRESUPUESTO
D1	Comercialización	10M
D2	Desarrollo	12M

Proyectar:

DEPTOs sobre DEPTO#, PRESUPUESTO

Resultado:

DEPTO#	PRESUPUESTO
D1	10M
D2	**12M**
D3	5M

Juntar:

DEPTOs y EMP; sobre DEPTO#

Resultado:

DEPTO#	NOMDEPTO	PRESUPUESTO	EMP#	NOMEMP	SALARIO
D1	Comercialización	10M	E1	López	**40K**
D1	Comercialización	10M	E2	Cheng	**42K**
D2	Desarrollo	12M	E3	Pérez	**30K**
D2	Desarrollo	12M	E4	Hernández	**35K**

Figura 3.2 Operaciones restringir, proyectar y juntar (ejemplos).

De los tres ejemplos, el único que puede necesitar de una mayor explicación es el ú el ejemplo de juntar. Observe primero que, en efecto, las dos tablas DEPTO y EMP tienen de hecho una columna en común, DEPTO#; de modo que pueden ser juntadas con base en los \ alo res comunes de esa columna. Una fila dada de la tabla DEPTO será combinada con una fila dad; de la tabla EMP (para producir una fila de la tabla de resultado) únicamente si las dos filas cuestión tienen un valor común para DEPTO#. Por ejemplo, las filas DEPTO y EMP

DEPTO#	NOMDEPTO	PRESUPUESTO
D1	Comercialización	10M

EMP#	NOMEMP	DEPTO#	SALARIO
E1	López	D1	40K

(los nombres de columna son mostrados por claridad) se combinan para producir la fila re sultante

DEPTO#	NOMDEPTO	PRESUPUESTO	EMP#	NOMEMP	SALARIO
D1	Comercialización	10M	E1	López	40K

ya que tienen el mismo valor, D1, en la columna común. Observe que el valor común ap¡ una sola vez y no dos, en la fila resultante. El resultado global de la junta contiene todas las fil posibles que pueden ser obtenidas de esta manera (y ninguna otra fila). En particular, ob¡ que debido a que ninguna fila de EMP tiene un valor D3 en DEPTO# (es decir, ningún empleado está actualmente asignado a ese departamento), no aparece en el resultado una fila para D3. aun cuando *sí* existe una fila para D3 en la tabla DEPTO.

Ahora bien, una idea que la figura 3.2 muestra claramente es que *el resultado de cada una de las tres operaciones es otra tabla* (en otras palabras, los operadores son en efecto "operadores que derivan tablas a partir de tablas", como mencioné anteriormente). Ésta es la propiedad de **cierre** de los sistemas relacionales y es muy importante. Básicamente, debido a que la salid de cualquier operación es el mismo tipo de objeto que la entrada —todas son tablas— **entonces** *la salida de una operación puede convertirse en la entrada de otra.* Por lo tanto es posible, por ejemplo, tomar una proyección de una junta, o una junta de dos restricciones, o una restricción de una proyección, y así sucesivamente. En otras palabras, es posible escribir *expresiones anidadas;* es decir, expresiones en las que los propios operandos están representados por expresiones generales, en vez de simples nombres de tablas. Este hecho tiene a su vez numerosas consecuencias importantes, como veremos más adelante (tanto en este capítulo como en otros subsiguientes).

Por cierto, cuando decimos que la salida de cada operación es otra tabla, es importante entender que estamos hablando *desde un punto de vista conceptual.* No pretendemos decir que e sistema realmente tenga que materializar el resultado de cada operación individual en su totalidad. Por ejemplo, suponga que intentamos calcular una restricción de una junta. Entonces, tan pronto como se forma una fila dada de la junta, el sistema puede confrontar de inmediato esa fila con la condición de restricción especificada para ver si pertenece al resultado final, y descartarla de inmediato de no ser así. En otras palabras, el resultado intermedio, que es la salida de la junta. no podría existir como una tabla completamente materializada. De hecho, como regla general. el sistema procura en gran medida *no* materializar en su totalidad los resultados intermedios por razones obvias de rendimiento. *Nota:* Si los resultados intermedios son materializados en su totalidad, a la estrategia de evaluación general de la expresión se le denomina (en forma nada sor

prendente) *evaluación materializada;* si los resultados intermedios son cedidos poco a poco a las operaciones subsiguientes, se le llama evaluación *canalizada.*

Otra idea ilustrada claramente por la figura 3.2 es que todas las operaciones se realizan un **conjunto a la** vez, no una fila a la vez; esto significa que los operandos y resultados son tablas completas en vez de sólo filas individuales, y que las tablas contienen *conjuntos* de filas. (Por supuesto, una tabla que contiene un conjunto de una sola fila es legal, como lo es también una tabla *vacía,* es decir, una tabla que no contiene fila alguna.) Por ejemplo, la junta de la figura 3.2 opera sobre dos tablas de tres y cuatro filas respectivamente, y devuelve una tabla resultante de cuatro filas. En contraste, las operaciones en los sistemas no relacionales se encuentran generalmente en el nivel de una fila a la vez o un registro a la vez; de ahí que esta *capacidad de procesamiento de conjuntos* sea una de las principales características distintivas de los sistemas relacionales (en la sección 3.5 a continuación, encontrará una explicación más amplia).

Regresemos por un momento a la figura 3.1. Hay un par de puntos adicionales con relación a la base de datos de ejemplo de esa figura:

- Primero, observe que los sistemas relacionales sólo requieren que la base de datos sea *percibida por el usuario* en forma de tablas. Las tablas son la estructura **lógica** en un sis tema relacional, no la estructura física. De hecho, en el nivel físico el sistema es libre de almacenar los datos en cualquier forma en que desee —utilizando archivos secuenciales, indexación, dispersión, cadenas de apuntadores, compresión, etcétera— con tal de que pueda asociar la representación almacenada con tablas en el nivel lógico. En otras pala bras, las tablas representan una *abstracción* de la forma en que los datos están almace nados físicamente; una abstracción en la cual *se ocultan al usuario* diversos detalles del nivel de almacenamiento, como la ubicación de los registros almacenados, la secuencia de los registros almacenados, las representaciones de los valores de los datos almacenados, los prefijos de registros almacenados, las estructuras de acceso almacenadas (como los índices), etcétera.

  Por cierto, el término "estructura lógica" en el párrafo anterior pretende abarcar los niveles tanto conceptual como externo, en términos de ANSI/SPARC. La idea es que —co mo expliqué en el capítulo 2— en un sistema relacional los niveles conceptual y externo serán relacionales, pero el nivel físico o interno no lo será. De hecho, la teoría relacional como tal no tiene absolutamente nada que decir acerca del nivel interno; de nuevo, se ocupa de cómo luce la base de datos ante el *usuario.* * El único requerimiento es que cualquiera que sea la estructura elegida, deberá implementar por completo la estructura lógica.

- Segundo, las bases de datos relacionales acatan un principio interesante, denominado **Prin cipio de Información:** *Todo el contenido de información de la base de datos está repre sentado en una y sólo una forma; es decir, como valores explícitos dentro de posiciones de columnas dentro de filas dentro de tablas.* Este método de representación es el *único* método disponible (por supuesto, en el nivel lógico) en un sistema relacional. En particular, **no** hay

---

* Es un hecho desafortunado que la mayoría de los productos SQL actuales no manejen en forma adecuada este aspecto de la teoría. Para ser más específicos, por lo regular sólo manejan transformaciones conceptuales/internas más bien restrictivas (por lo regular transforman directamente una tabla lógica en un archivo almacenado). Como consecuencia, no proporcionan tanta independencia física de datos como la que en teoría es capaz de proporcionar la tecnología relacional.

apuntadores que conecten una tabla con otra. Por ejemplo, en la figura 3.1 hay una conexión entre la fila D1 de la tabla DEPTO y la fila El de la tabla EMP, ya que el empleado El trabaja en el departamento D1; pero esa conexión no está representada por un apunta dor, sino por la aparición del valor D1 en la posición DEPTO# de la fila de EMP para E En contraste, en los sistemas no relacionales (por ejemplo IMS o IDMS), dicha información se representa por lo regular —como mencioné en el capítulo 1— por medio de algún tipo de *apuntador* que sea visible de manera explícita para el usuario.

Nota: Cuando decimos que no hay apuntadores en una base de datos relacional no queremos decir que no pueda haber apuntadores en el *nivel físico;* al contrario, en realidad puede haberlos y de hecho casi es seguro que los haya. Aunque, nuevamente, en un sistema relacional todos estos detalles de almacenamiento físico quedan ocultos ante el usuario.

3.3 R

Es suficiente por lo que respecta a los aspectos estructural y de manipulación del modelo relacional; ahora pasaremos al aspecto de la integridad. Considere una vez más la base de datos de departamentos y empleados de la figura 3.1. En la práctica podría requerirse que esa base de datos cumpliera cualquier cantidad de restricciones de integridad; por ejemplo, que los salario de los empleados tuvieran que estar (digamos) en un rango de 25K a 95K, los presupuestos de los departamentos tuvieran que estar (por decir algo) en el rango de 1M a 15M, etcétera. Sil bargo, algunas de estas restricciones son de una importancia pragmática tal, que gozan de una nomenclatura especial. Para ser más específicos:

1. Cada fila de la tabla DEPTO debe incluir un valor DEPTO# único; en forma similar, cada fila de la tabla EMP debe incluir un valor de EMP# único. En términos generales, decimos que las columnas DEPTO# de la tabla DEPTO y EMP# de la tabla EMP son las **claves pr marias** de sus respectivas tablas. (Recuerde que en las figuras del capítulo 1 señalamos las claves primarias mediante un doble subrayado.)
2. Cada valor DEPTO# de la tabla EMP debe existir como un valor DEPTO# en la tabla DEPTO, para reflejar el hecho de que cada empleado debe estar asignado a un departamento existente. En términos generales, decimos que la columna DEPTO# de la tabla EMP es una clave **externa** que hace referencia a la clave primaria de la tabla DEPTO.

Cerramos esta sección con una definición del modelo relacional para fines de futuras re ferencias (no obstante que la definición es bastante abstracta y no será muy comprensible en es etapa). Para ser breves, **el modelo relacional** consta de los siguientes cinco componentes:

1. Un conjunto abierto de **tipos escalares** (incluyendo en particular el tipo *lógico* o ***valor verdadero);***
2. **Un generador de tipos de relación** y una interpretación propuesta de dichos tipos;
3. Herramientas para definir **variables de relación** de dichos tipos de relación generados;
4. Una operación **asignación relacional** para asignar valores de relación a las variables de relación mencionadas;
5. Un conjunto abierto de **operadores relacionales** genéricos para derivar valores de relación de otros valores de relación.

Como puede ver, el modelo relacional es mucho más que sólo "tablas con restringir, proyectar y juntar", aunque a menudo se le caracteriza informalmente de esa manera.

Nota: Tal vez le sorprendió que no mencionáramos explícitamente a las restricciones de integridad en la definición anterior. Sin embargo, el hecho es que dichas restricciones representan sólo una aplicación (aunque una muy importante) de los operadores relacionales; esto es, que dichas restricciones se formulan —en todos los casos de manera conceptual— en términos de dichos operadores, como veremos en el capítulo 8.

3.3 RELACIONES Y VARIABLES DE RELACIÓN

Si es cierto que una base de datos relacional es en esencia sólo una base de datos en la que los datos son percibidos como tablas —y, por supuesto, esto *es* cierto—, entonces una buena pregunta es: ¿exactamente por qué llamamos relacional a dicha base de datos? La respuesta es sencilla (de hecho, la mencionamos en el capítulo 1): "Relación" es sólo un término matemático para una tabla; para ser precisos, una tabla de cierto tipo específico (los detalles se expondrán en el capítulo 5). De ahí que, por ejemplo, podamos decir que la base de datos de departamentos y empleados de la figura 3.1 contiene dos *relaciones.*

Ahora, en contextos informales es común tratar los términos "relación" y "tabla" como si fueran sinónimos; de hecho, el término "tabla" se usa informalmente con mucha más frecuencia que el término "relación". Pero vale la pena dedicar un momento a comprender por qué se introdujo en primer lugar el término "relación". En resumen, la explicación es la siguiente:

- Como hemos visto, los sistemas relacionales se basan en el modelo relacional. A su vez, este modelo es una teoría abstracta de datos que está basada en ciertos aspectos de las matemáti cas (principalmente en la teoría de conjuntos y la lógica de predicados).
- Los principios del modelo relacional fueron establecidos originalmente en 1969-70 por E. F. Codd (en ese entonces, investigador de IBM). Fue a fines de 1968 que Codd, matemático de formación, descubrió por primera vez que la disciplina de las matemáticas podía ser usada para dar ciertos principios sólidos y cierto rigor a un campo —la administración de base de datos— que hasta ese momento era muy deficiente en cualquiera de estas cuali dades. En un principio, las ideas de Codd se difundieron ampliamente en un artículo que hoy en día es clásico: "A Relational Model of Data for Large Shared Data Banks" (vea la referencia [5.1] del capítulo 5).
- Desde entonces, esas ideas —ahora aceptadas casi en forma universal— han tenido gran in fluencia en prácticamente cada uno de los aspectos de la tecnología de base de datos; y de hecho también en otros campos, como los de la inteligencia artificial, el procesamiento del lenguaje natural y el diseño de sistemas de hardware.

Ahora bien, el modelo relacional como lo formuló originalmente Codd hizo utilizó deliberadamente ciertos términos (como el propio término "relación") que en ese momento no eran familiares en los círculos de tecnología de la información (aunque en algunos casos los conceptos sí lo eran). El problema fue que muchos de los términos más conocidos eran muy *confusos,* carecían de la precisión necesaria para una teoría formal como la que Codd proponía. Por ejemplo, considere el término "registro". En diferentes momentos y en distintos contextos este término puede significar ya sea la *ocurrencia* de un registro o un *tipo* de registro; un registro *lógico* o un registro *físico*; un registro *almacenado* o un registro *virtual,* y quizás también otras cosas. Por lo tanto, el modelo relacional no emplea en absoluto el término "registro"; en su lugar usa

el término "tupia", al que es posible asignar una definición muy precisa. Aquí no damos esa definición ya que para los fines actuales, es suficiente decir que el término "tupia" corresponde aproximadamente a la noción de una fila (tal como el término "relación" corresponde aproximadamente a la noción de una tabla). Cuando comencemos a estudiar los aspectos más formales de los sistemas relacionales (en la parte II), haremos uso de la terminología formal, pero en este capítulo no pretendemos ser tan formales (al menos no en su mayoría) y nos apegaremos prin cipalmente a términos como "fila" y "columna" que son razonablemente familiares. Sin embargo, un término formal que *sí* usaremos mucho a partir de este punto es el término "relación" Regresemos una vez más a la base de datos de departamentos y empleados de la figura 3.1 para hacer otro señalamiento importante. El hecho es que DEPTO y EMP en esa base de datos son en realidad *variables* de relación; es decir, variables cuyos valores son *valores* de relación (diferentes valores de relación en diferentes momentos). Por ejemplo, suponga que EMP tiene actualmente el valor —es decir, el valor de *relación*— que se muestra en la figura 3.1 y suponga que eliminamos la fila de Hernández (el empleado número E4):

```
DELETE EMP WHERE EMP# = 'E4' ;
```

El resultado aparece en la figura 3.3.

EMP

EMP#	NOMEMP	DEPTO#	SALARIO
E1	López	D1	40K
E2	Cheng	D1	42K
E3	Pérez	D2	30K

Figura 3.3 La variable de relación EMP después de eliminar la fila E4.

De manera conceptual, lo que sucedió aquí es que el *valor de relación anterior de EMP fue reemplazado* en bloque *por un valor de relación completamente nuevo.* Desde luego, el valor anterior (con cuatro filas) y el nuevo (con tres) son muy similares, pero de manera conceptual *son* valores diferentes. De hecho la operación de eliminación en cuestión es básicamente una forma abreviada de una cierta operación de **asignación relacional** que podría ser como la siguiente:

```
EMP := EMP MINUS ( EMP WHERE EMP# • ' E4' ) ;
```

{Nota: Tanto el DELETE original como la instrucción de asignación equivalente están expresadas en un lenguaje denominado **Tutorial D** [3.3]. MINUS es la palabra reservada de **Tutorial D** para la operación de *diferencia* relacional; vea el capítulo 6). Como en todas las asignaciones, lo que sucede aquí, hablando en forma conceptual, es que (a) se evalúa la expresión de la derecha y luego (b) el resultado de la evaluación se asigna a la variable de la izquierda (por supuesto, la parte izquierda debe por definición identificar específicamente a una *variable).* Por lo tanto, como ya mencionamos, el efecto neto es reemplazar el valor "anterior" de EMP por uno "nuevo".

Desde luego, las operaciones relacionales INSERT y UPDATE también son en esencia formas abreviadas de ciertas asignaciones relacionales.

Ahora bien, desgraciadamente gran parte de la literatura usa el término *relación* cuando en realidad se refiere a una *variable* de relación (al igual que cuando se refiere a una relación *como tal;* es decir, a un *valor* de relación). Sin embargo, esta práctica ha llevado ciertamente a una confusión. Por lo tanto, a lo largo de este libro, haremos una cuidadosa distinción entre las variables de relación y las relaciones *como tales;* de hecho, de acuerdo con la referencia [3.3], usaremos el término **varrel** como una abreviatura conveniente de *variable de relación* (o *variable relacional)* y tendremos cuidado de enunciar nuestras observaciones en términos de varrels (y no de relaciones) cuando realmente nos refiramos a las variables relacionales.

Nota: Debo advertirle que el término *varrel* no es de uso común; aunque debería serlo. En realidad creemos que es importante aclarar la diferencia entre las relaciones *como tales* (es decir, los valores de relación) y las variables de relación, aunque en ediciones previas de este libro no se haya hecho y aunque la mayoría de la literatura actual de base de datos siga sin hacerlo.

QUE SIGNIFICAN LAS RELACIONES

En el capítulo 1, mencioné el hecho de que, en las relaciones, las columnas tienen **tipos de datos** asociados.* Y al final de la sección 3.2, dije que el modelo relacional incluye un "conjunto abierto de tipos [de datos]". Lo que esto significa (entre otras cosas) es que **los usuarios podrán** definir **sus propios tipos** y desde luego, también usar los tipos definidos por el sistema (o *integrados).* Por ejemplo, podríamos definir los tipos de la siguiente manera (de nuevo la sintaxis de **Tutorial D;** los puntos suspensivos "..." representan parte de las definiciones que no son aplicables a la explicación actual):

```
TYPE EMP# ... ;
TYPE NOMBRE . . . ;
TYPE DEPTO# ... ;
TYPE DINERO ... ;
```

Por ejemplo, el tipo EMP# puede ser visto como *el conjunto de todos los números de empleado posibles;* el tipo NOMBRE como *el conjunto de todos los nombres posibles;* y así sucesivamente.

Ahora considere la figura 3.4, que es básicamente la parte EMP de la figura 3.1, ampliada para mostrar los tipos de datos de las columnas. Como la figura indica, cada relación —para ser más precisos, cada *valor* de relación— tiene dos partes, un conjunto de parejas nombre-de-columna:nombre-de-tipo (el **encabezado)** y un conjunto de filas que se apegan a ese encabezado (el **cuerpo).** *Nota:* En la práctica, a menudo ignoramos los componentes del encabezado nombre-de-tipo (como en efecto lo hemos hecho en ejemplos anteriores), pero debe entenderse que siempre están ahí conceptualmente.

Ahora bien, existe una forma de pensar muy importante (aunque tal vez no muy común) acerca de las relaciones, y es como sigue:

---

*E1 término relacional más usual para los tipos de datos es dominios, como veremos en el capítulo 5.

EMP

EMP# : EMP#	NOMEMP : NOMBRE	DEPTO# : DEPTO#	SALARIO : DINERO
E1	López Cheng	D1	40K
E2	Pérez	D1	42K
E3	Hernández	D2	30K
E4		D2	35K

Figura 3.4 Ejemplo del valor de relación EMP, que muestra los tipos de columnas.

- Primero, dada una relación *r*, el encabezado de *r* denota un cierto **predicado** o función valuada como verdadera;
- Segundo, como mencioné brevemente en el capítulo 1, cada fila en el cuerpo de *r* denota una cierta **proposición verdadera,** obtenida del predicado por medio de la sustitucic ciertos valores de *argumento* del tipo apropiado en los *indicadores de posición o paráme-*
  tros de ese predicado ("creando un ejemplar del predicado").

Por ejemplo, en el caso de la figura 3.4, el predicado luce como lo siguiente:

El empleado EMP# se llama NOMEMP, trabaja en el departamento DEPTO# y gana salario de SALARIO

(los parámetros son EMP#, NOMEMP, DEPTO# y SALARIO, que corresponden por sup a las cuatro columnas de EMP). Y las proposiciones verdaderas correspondientes son:

*El empleado El se llama López, trabaja en el departamento D1 y gana el salario **40K***

(proposición que se obtuvo al sustituir los parámetros correspondientes por el valor El, de **tipo** EMP#; el valor López, de tipo NOMBRE; el valor DI, de tipo DEPTO#; y el valor 40K. de tipo DINERO);

*El empleado E2 se llama Cheng, trabaja en el departamento DI y gana el salario **42K***

(proposición que se obtuvo al sustituir los parámetros correspondientes por el valor E2. de t EMP#; el valor Cheng, de tipo NOMBRE; el valor D1, de tipo DEPTO# y el valor 42K. de tipo DINERO); y así sucesivamente. Por lo tanto, resumiendo:

- **Los *tipos* son (conjuntos de) cosas de las que podemos hablar;**
- **Las *relaciones* son (conjuntos de) cosas que decimos acerca de las cosas de las que podemos hablar.**

(Hay una analogía que podría ayudarle a apreciar y recordar estos puntos importantes: ***Los tipo son a las relaciones lo que los sustantivos son a las oraciones.)*** Por lo tanto, en el ejemplo, las cosas de las que podemos hablar son los números de empleado, los nombres, los números de de-parlamento y los valores de dinero, mientras que las cosas que decimos son expresiones ver-daderas de la forma "El empleado con el número de empleado especificado tiene el n especificado, trabaja en el departamento especificado y gana el salario especificado".

3.5

De lo anterior se desprende que:

- Primero, tanto los tipos como las relaciones son *necesarios* (sin tipos, no tenemos nada de qué hablar; sin relaciones, no podemos decir nada).
- Segundo, los tipos y las relaciones son *suficientes*, así como necesarios; es decir, no necesitamos nada más, hablando de manera lógica.

Nota: También se desprende que los *tipos y las relaciones no son la misma cosa.* Desafortunadamente, ciertos productos comerciales —¡por definición, no los relacionales!— están confundidos en este punto. Abordaremos esta confusión en el capítulo 25 (sección 25.2).

Por cierto, es importante entender que *toda* relación tiene un predicado asociado; incluyendo las relaciones que se derivan de otras mediante operadores relaciónales como el de juntar. Por ejemplo, la relación DEPTO de la figura 3.1 y las tres relaciones resultantes de la figura 3.2 tienen los siguientes predicados:

- *DEPTO:* "El departamento DEPTO# se llama NOMDEPTO y tiene un presupuesto PRESUPUESTO".
- *Restricción de DEPTO donde PRESUPUESTO > 8M:* "El departamento DEPTO# se llama NOMDEPTO y tiene un presupuesto PRESUPUESTO, el cual es mayor a ocho millones de dólares".
- *Proyección de DEPTO sobre DEPTO# y PRESUPUESTO:* "El departamento DEPTO# tiene algún nombre y tiene un presupuesto PRESUPUESTO".
- *Junta de DEPTO y EMP sobre DEPTO#:* "El departamento DEPTO# se llama NOMDEPTO y tiene un presupuesto PRESUPUESTO y el empleado EMP# se llama NOMEMP, trabaja en el departamento DEPTO# y gana un salario de SALARIO". Observe que este predicado tiene seis parámetros, no siete; las dos referencias a DEPTO# denotan el mismo parámetro.

3.5 OPTIMIZACION

Como expliqué en la sección 3.2, todas las operaciones relacionales tales como restringir, proyectar y juntar son operaciones *en el nivel de conjunto.* Como consecuencia, se dice a menudo que los lenguajes relacionales *no son de procedimientos,* en el sentido de que los usuarios especifican *el qué,* no *el cómo;* es decir, dicen lo que desean sin especificar un procedimiento para obtenerlo. El proceso de "navegar" por los datos a fin de satisfacer la petición del usuario es realizado automáticamente por el sistema, en vez de ser realizado en forma manual por el usuario. Por esta razón, en ocasiones se dice que los sistemas relacionales realizan una **navegación automática.** En contraste, en los sistemas no relacionales la navegación es generalmente responsabilidad del usuario. La figura 3.5 muestra una ilustración impactante de los beneficios de la navegación automática y compara una instrucción INSERT de SQL con el código de "navegación manual" que tendría que escribir el usuario para lograr el efecto equivalente en un sistema no relacional (de hecho un sistema de red CODASYL; el ejemplo se tomó de la referencia [1.5] del capítulo sobre bases de datos de red). *Nota:* La base de datos es la base de datos de proveedores y partes que ya conocemos. Para una mayor explicación, consulte la sección 3.9.

A pesar de las observaciones del párrafo anterior, debemos decir que, aunque sea común, el término *no procedimientos* no es muy satisfactorio, ya que las categorías de procedimientos

```
INSERT INTO VP ( V#, P#, CANT )
       VALUES ( 'V4', 'P3', 1000 )
```

```
MOVE 'V4' TO V# IN V
FIND CALC V
ACEPT V-VP-ADDR FROM V-VP CURRENCY
FIND LAST VP WITHIN V-VP
while VP found PERFORM
   ACCEPT V-VP-ADDR FROM V-VP CURRENCY
   FIND OWNER WITHIN V-VP
   GET P
   IF P# IN P < 'P3'
       leave loop
   END-IF
   FIND PRIOR VP WITHIN V-VP
END-PERFORM MOVE 'P3' TO P# IN
P FIND CALC P
ACCEPT P-VP-ADDR FROM P-VP CURRENCY
FIND LAST VP WITTHIN P-VP while VP
found PERFORM
   ACCEPT P-VP-ADDR FROM P-VP CURRENCY
   FIND OWNER WITHIN V-VP
   GET V
   IF V# IN V < 'V4'
       leave loop
   END-IF
   FIND PRIOR VP WITHIN P-VP
END-PERFORM
MOVE 1000 TO CANT IN VP
FIND DB-KEY IS V-VP-ADDR
FIND DB-KEY IS P-VP-ADDR
STORE VP
CONNECT VP TO V-VP
CONNECT VP TO P-VP
```

Figura 3.5 Navegación automática frente a navegación manual.

o de no procedimientos no son absolutas. Lo mejor que podemos decir es que un lenguaje A es en mayor o menor medida relativo a los procedimientos que otro lenguaje *B.* Quizás una mejor de poner las cosas sería decir que los lenguajes como SQL están *en un nivel más alto abstracción* que los lenguajes como C++ y COBOL (o que los sublenguajes de datos como lo: que se encuentran por lo regular en los DBMSs no relacionales, como muestra la figura 3 esencia, es esta elevación del nivel de abstracción la responsable del aumento en la productivi-dad que pueden ofrecer los sistemas relacionales.

Decidir cómo realizar la navegación automática a la que nos referimos anteriormente responsabilidad de un componente muy importante del DBMS denominado **optimizador** (men-cionamos brevemente este componente en el capítulo 2). En otras palabras, para cada pe relacional del usuario, es trabajo del optimizador seleccionar una forma eficiente de implemen-tar esa petición. A manera de ejemplo, supongamos que el usuario emite la siguiente pe (una vez más en **Tutorial D):**

```
RESULT := ( EMP WHERE EMP# - 'E4' ) { SALARIO }
```

Explicación: La expresión entre paréntesis ("EMP WHERE ...") denota una restricción del valor actual de la varrel EMP solamente para la fila en la que EMP# es E4. Entonces, el nombre de columna entre llaves ("SALARIO") hace que el resultado de esa restricción se proyecte sobre la columna SALARIO. Por último, la operación de asignación (":=") hace que el resultado de esa proyección sea asignado a la varrel RESULT. Después de esa asignación, RESULT contiene una relación de una sola columna y una sola fila que contiene el salario del empleado E4. (Por cierto, observe que en este ejemplo estamos haciendo uso implícitamente de la propiedad relacional de *cierre;* hemos escrito en la parte derecha una expresión relacional anidada, en la que la entrada a la proyección es la salida de la restricción.)

Ahora, incluso en este ejemplo sencillo, existen por lo menos dos formas posibles de realizar el acceso de datos necesario:

1. Mediante un examen físico secuencial de (la versión almacenada de) la varrel EMP, hasta encontrar los datos requeridos;
2. Si existe un índice en (la versión almacenada de) la columna EMP# —lo que es probable que ocurra en la práctica, ya que se supone que los valores de EMP# son únicos y que muchos sistemas de hecho *requieren* de un índice a fin de hacer cumplir esta característica de valor único— se utiliza entonces ese índice y se pasa por lo tanto directamente a los datos requeridos.

El optimizador elegirá cuál de estas dos estrategias adoptar. De manera más general, dada una petición en particular, el optimizador elegirá la estrategia para implementar esa petición basándose en consideraciones como las siguientes:

- A qué varrels se hace referencia en la petición;
- Qué tan grandes son actualmente esas varrels;
- Qué índices existen;
- Qué tan selectivos son esos índices;
- Cómo están agrupados físicamente los datos en el disco;
- Qué operadores relacionales están involucrados;

entre otras. Por lo tanto, para repetir: Los usuarios sólo especifican qué datos desean, no cómo obtenerlos; la estrategia de acceso para obtener los datos es seleccionada por el optimizador ("navegación automática"). Los usuarios y los programas de usuario son independientes de dichas estrategias de acceso, lo que desde luego es esencial si se pretende lograr la independencia de los datos.

Tenemos mucho más qué decir acerca del optimizador en el capítulo 17.

3.6 EL CATALOGO

Como expliqué en el capítulo 2, todo DBMS debe proporcionar una función de **catálogo** o **diccionario.** El catálogo es el lugar donde —entre otras cosas— se guardan los diversos esquemas (externo, conceptual, interno) y todas las transformaciones correspondientes (externa/conceptual, conceptual/interna). En otras palabras, el catálogo contiene información detallada (a veces denominada **información de descriptores** o **metadatos)** respecto de los distintos objetos que

son de interés para el propio sistema. Ejemplos de dichos objetos son las varrels, los índices, los usuarios, las restricciones de integridad, las restricciones de seguridad, etcétera. La información de descriptores es esencial para que el sistema haga bien su trabajo. Por ejemplo, el optimizador utiliza información del catálogo relacionada con los índices y otras estructuras de almacenamiento físico (así como otro tipo de información) para poder decidir cómo implementar las peticiones del usuario. En forma similar, el subsistema de seguridad utiliza información del catálogo relacionada con los usuarios y las restricciones de seguridad (vea el capítulo 16), principalmente para autorizar o negar dichas peticiones.

Ahora, una característica atractiva de los sistemas relacionales es que en ellos el *propio catálogo consiste en varrels* (para ser más precisos, varrels *del sistema,* llamadas así para distinguirlas de las normales de usuario). Como resultado, los usuarios pueden consultar el catálogo exactamente de la misma manera en que consultan sus propios datos. Por ejemplo, el catálogo comprenderá por lo regular dos varrels de sistema denominadas TABLA y COLUMNA, cuya finalidad es describir las tablas (es decir, las varrels) de la base de datos y las columnas de dichas tablas. (Decimos "por lo regular" debido a que el catálogo no es igual en todos los sistemas; las diferencias surgen debido a que el catálogo de un sistema en particular contiene una gran cantidad de información específica de ese sistema). Para la base de datos de departamentos y empleados de la figura 3.1, las varrels TABLA y COLUMNA podrían lucir —a manera de bosquejo- como se muestra en la figura 3.6.*

Nota: Como mencioné en el capítulo 2, el catálogo normalmente debe describirse a sí mismo; es decir, debe incluir entradas que describan las propias varrels del catálogo. Sin embargo. dichas entradas no aparecen en la figura 3.6. Vea el ejercicio 3.3 al final del capítulo.

TABLA

NOMTABLA	CONTCOL	CONTFILA	
DEPTO	3	3	.....
EMP	4	4	

COLUMNA

NOMTABLA	NOMCOL	
DEPTO	DEPTO	
DEPTO	NOMDEPTO	
DEPTO	PRESUPUESTO	
EMP	EMP#	
EMP	NOMEMP	
EMP	DEPT0#	
EMP	SALARIO	

Figura 3.6 Catálogo (bosquejo) de la base de datos de departamentos y empleado

*Observe que la presencia de una columna CONTFILA en la figura, sugiere que las operaciones de inserción y eliminación (INSERT y DELETE) de la base de datos generarán una actualización del catálogo como un efecto colateral. En la práctica, CONTFILA podría ser actualizada sólo a petición (por ejemplo, a! ejecutar alguna utilería), lo que significa que los valores podrían no ser siempre los actuales.

Suponga ahora que un usuario de la base de datos de departamentos y empleados desea saber exactamente qué columnas contiene la varrel DEPTO (obviamente estamos dando por hecho que por alguna razón el usuario no tiene todavía esta información). Entonces, la expresión

```
( COLUMNA WHERE NOMTABLA = 'DEPTO' )  { NOMCOL }
```

realiza el trabajo. *Nota:* Si hubiésemos querido conservar el resultado de esta consulta de alguna forma más permanente, podríamos haber asignado el valor de la expresión a alguna otra varrel, como en el ejemplo de la sección anterior. En la mayoría de nuestros ejemplos restantes omitiremos este paso final de asignación (tanto en éste como en los capítulos subsiguientes). Aquí tenemos otro ejemplo: "¿Qué varrels incluyen una columna de nombre EMP#?"

```
( COLUMNA WHERE NOMCOL = ' EMP#' )  { NOMTABLA }
```

Ejercicio: ¿Qué es lo que hace lo siguiente?

```
(  ( TABLA JOIN COLUMNA )
          WHERE CONTCOL < 5 ) { NOMTABLA, NOMCOL }
```

3.7 VARIABLES DE RELACIÓN BASE Y VISTAS

Hemos visto que a partir de un conjunto de varrels como DEPTO y EMP, y un conjunto de valores de relación de éstas, las expresiones relacionales nos permiten obtener otros valores de relación a partir de los ya dados (por ejemplo, juntando dos de las varrels dadas). Ahora, es momento de presentar un poco más de terminología. A las varrels originales (dadas) se les denomina **varrels base** y a sus valores de relación se les llama **relaciones base;** a una relación que es o que puede ser obtenida a partir de dichas relaciones base por medio de alguna expresión relacional, se le denomina relación **derivada** o **derivable.** *Nota:* En la referencia [3.3], a las varrels base se les llama varrels *auténticas.*

Ahora bien, en primer lugar, los sistemas relacionales tienen obviamente que proporcionar un medio para crear las varrels base. Por ejemplo, en SQL esta tarea es realizada por medio de la instrucción CREATE TABLE (aquí, "TABLE" tiene un significado muy específico: el de varrel base). Y a estas varrels base obviamente se les tiene que *dar un nombre;* por ejemplo:

```
CREATE TABLE EMP . . . ;
```

Sin embargo, por lo regular los sistemas relacionales también manejan otro tipo de varrel con nombre, denominada **vista,** cuyo valor en cualquier momento dado es una relación *derivada* (de donde se puede imaginar una vista, a grandes rasgos, como una **varrel derivada).** El valor de una vista determinada en un momento dado, es cualquiera que sea el resultado de evaluar cierta expresión relacional en ese momento; dicha expresión relacional es especificada en el momento de que se crea la vista en cuestión. Por ejemplo, la instrucción

```
CREATE VIEW EMPSUP AS
     ( EMP WHERE SALARIO > 33K )  { EMP#, NOMEMP, SALARIO } ;
```

podría ser usada para definir una vista llamada EMPSUP. *Nota:* Por razones que no son relevantes en este momento, este ejemplo está expresado en una combinación de SQL y **Tutorial D.**

Cuando esta instrucción es ejecutada, la expresión relacional que sigue a AS —la **expresión que define la vista**— no es evaluada, sino que el sistema simplemente la "recuerda" de alguna manera (de hecho, guardándola en el catálogo bajo el nombre especificado EMPSUP). Sin embargo, para el usuario, ahora es como si en realidad fuese una varrel de la base de datos, denominada EMPSUP, con un valor actual como se indica (sólo) en las partes sin sombrear de la figura 3.7. Y el usuario debe poder operar sobre esa vista tal como si fuera una varrel base. *Nota:* Si (como sugerí anteriormente) pensamos en DEPTO y en EMP como varrels *auténticas,* entonces podríamos pensar en EMPSUP como una varrel *virtual;* es decir, una varrel que aparentemente existe por derecho propio, pero que de hecho no existe (su valor, en cualquier momento dado, depende de los valores de otras vareéis determinadas).

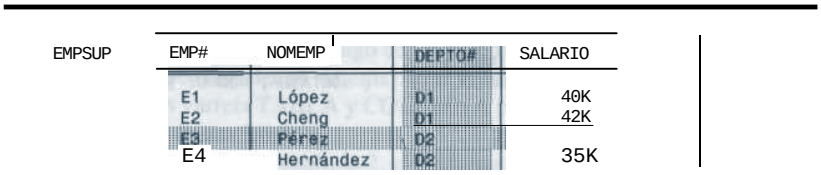


Figura 3.7 EMPSUP como una vista de EMP (partes sin sombrear).

Sin embargo, note detenidamente que aunque decimos que el valor de EMPSUP es la relación que resultaría si se evaluara la expresión de definición de la vista, no pretendemos sugerir que ahora tenemos una *copia independiente* de los datos; es decir, no pretendemos sugerir que *se evalúe* realmente la expresión de definición de la vista. Al contrario, la vista es en efecto sólo una *ventana* hacia el interior de la varrel base EMP. Como consecuencia, todo cambio a esa varrel subyacente será visible de manera automática e instantánea a través de esa ventana (suponiendo desde luego que estos cambios se localizan dentro de la parte no sombreada). En forma similar, los cambios a EMPSUP se aplicarán de manera automática e instantánea a la varrel EMP y por lo tanto, serán visibles a través de la ventana (vea un ejemplo más adelante).

El siguiente es un ejemplo de una operación de recuperación sobre la vista EMPSUP:

```
( EMPSUP WHERE SALARIO < 42K )  { EMP#, SALARIO }
```

Dados los datos de ejemplo de la figura 3.7, el resultado se verá como sigue:

EMP#	SALARIO
E1	40K
E4	35K

De manera conceptual, las operaciones frente a una vista, como la operación de recuperación que acabamos de mostrar, se manejan reemplazando las referencias para el nombre de

la vista por la expresión de definición de la vista (es decir, la expresión que fue guardada en el catálogo). Por lo tanto, en el ejemplo la expresión original

```
( EMPSUP WHERE SALARIO < 42K ) { EMP#, SALARIO }
```

es modificada por el sistema para convertirse en

```
( ( ( EMP WHERE SALARIO > 33K ) { EMP#, NOMEMP, SALARIO } )
          WHERE SALARIO < 42K )  { EMP#, SALARIO }
```

(pusimos en cursivas el nombre de la vista en la expresión original y el texto de reemplazo en la versión modificada). La expresión modificada puede entonces simplificarse a sólo

```
( EMP WHERE SALARIO > 33K AND SALARIO < 42K ) { EMP#, SALARIO }
```

(vea el capítulo 17), la cual cuando es evaluada produce el resultado mostrado anteriormente. En otras palabras, la operación original frente a la vista se convierte en una operación equivalente frente a la varrel base subyacente y entonces, dicha operación es ejecutada normalmente (para ser más precisos, es *optimizada y* ejecutada en la forma normal). A manera de otro ejemplo, considere la siguiente operación DELETE:

```
DELETE EMPSUP WHERE SALARIO < 42K ;
```

La operación de eliminación que en realidad se ejecuta es similar a la siguiente:

```
DELETE EMP WHERE SALARIO > 33K AND SALARIO < 42K ;
```

Ahora, la vista EMPSUP es muy sencilla, ya que consiste (hablando en términos generales) de un subconjunto de filas y columnas de una sola varrel base subyacente. Sin embargo, en principio una definición de vista puede ser *de una complejidad arbitraria,* ya que en esencia es sólo una expresión relacional con nombre (puede incluso referirse a otras vistas). Por ejemplo, aquí tenemos una vista cuya definición comprende una combinación de dos varrels subyacentes:

```
CREATE VIEW JOINEX AS
   ( ( EMP JOIN DEPTO ) WHERE PRESUPUESTO > 7M )      EMP#,
                                                     DEPT0#
```

En el capítulo 9 regresaremos a toda la cuestión de la definición y el procesamiento de vistas.

Por cierto, ahora podemos explicar la observación del capítulo 2 que está cerca del final de la sección 2.2, en el sentido de que el término "vista" tiene un significado más bien específico en el contexto relacional y que éste no es idéntico al significado que se le asigna en la arquitectura ANSI/SPARC. En el nivel externo de esa arquitectura, la base de datos se percibe como una "vista externa" definida por un esquema externo (donde diferentes usuarios pueden tener distintas vistas externas). En contraste, en los sistemas relacionales, una vista (como expliqué antes) es específicamente una varrel *con nombre, derivada* y *virtual.* Por lo tanto, la relación análoga de una "vista externa" de ANSI/SPARC es (por lo regular) un conjunto de diversas varrels, cada una de las cuales es una vista en el sentido relacional, y el "esquema externo" consiste en las definiciones de esas vistas. (De aquí que las vistas en el sentido relacional sean la forma

que tiene el modelo relacional de proporcionar la **independencia lógica de los datos;** aunque una vez más tenemos que decir que los productos comerciales actuales son deficientes en g medida a este respecto. Vea el capítulo 9.)

Ahora bien, la arquitectura ANSI/SPARC es bastante general y permite una variabilidad a bitraria entre los niveles externo y conceptual. En principio, aun los *tipos* de estructuras de datos soportados en los dos niveles podrían ser diferentes; por ejemplo, el nivel conceptual podría: relacional, mientras que un usuario dado podría tener una vista externa que fuese jerárquica. Sin embargo, en la práctica la mayoría de los sistemas usan el mismo tipo de estructura como base para ambos niveles, y los productos relacionales no son la excepción a esta regla; las vista siguen siendo varrels, como lo son las varrels base. Y puesto que se maneja el mismo tipo de objetos en ambos niveles, se aplica a éstos el mismo sublenguaje de datos (por lo regular SQL). En realidad, el hecho de que una vista sea una varrel es precisamente uno de los puntos fuertes de los sistemas relacionales; esto es importante exactamente en el mismo sentido en que, en matemáticas, un subconjunto es también un conjunto. *Nota:* Sin embargo, los productos SQL y el estándar de SQL (vea el capítulo 4) a menudo parecen ignorar este punto, ya que se refieren repetidamente a "tablas y vistas" (con la implicación de que una vista no es una tabla). Le aconsejamos seriamente *no* caer en esta trampa común de considerar que las "tablas" (o "varrels") sólo significan en forma específica, tablas (o varrels) *base*.

Es necesario hacer una observación final sobre el tema de las varrels frente a las vistas. La diferenciación entre ambas se caracteriza con frecuencia de la siguiente manera:

- Las varrels base "existen realmente", en el sentido de que representan datos que en reali dad están almacenados en la base de datos;
- En contraste, las vistas "no existen realmente", sino que sólo proporcionan diferentes formas de ver a "los datos reales".

Sin embargo, esta caracterización, aunque tal vez sea útil en un sentido informal, no refleja con precisión el verdadero estado de las cosas. Es cierto que los usuarios pueden *concebir* las **varrels** base como si estuvieran almacenadas físicamente; de hecho, en cierta forma la idea general de los sistemas relacionales es permitir a los usuarios que conciban las varrels base como si existieran físicamente, sin tener que ocuparse de cómo dichas variables están representadas realmente en el almacenamiento. Pero —¡y es un gran pero!— esta forma de pensar no debe interpretarse como que una varrel base está físicamente almacenada en cualquier clase de for directa (por ejemplo, en un solo archivo almacenado).* Como expliqué en la sección 3.2, las varrels base se conciben mejor como una *abstracción* de algún conjunto de datos almacenados; una abstracción en la que están ocultos todos los detalles del nivel de almacenamiento, En principio, puede haber un grado arbitrario de diferenciación entre una varrel base y su contraparte almacenada.Un ejemplo sencillo podría ayudar a aclarar esta idea. Considere una vez más la base de datos de departamentos y empleados. La mayoría de los sistemas relacionales actuales, probablemente implementarían la base de datos con dos archivos almacenados, uno para cada una de las varrels

---

*La siguiente cita de un libro reciente muestra varias de las confusiones que hemos venido exponiendo, asi como otras que expusimos antes en la sección 3.3:"...es importante distinguir entre las relaciones almacenadas, que son *tablas*, y las relaciones virtuales, que son *vistas* . . . usaremos *relación* sólo donde pueda usarse una tabla o una vista. Cuando queramos enfatizar que una relación está almacenada, en vez de una vista, usaremos ocasionalmente el término *relación base* o *tabla base."* Lamentablemente, esta **cita** no rara en lo absoluto.

base. Pero no existe en absoluto ninguna razón lógica por la que no pueda tratarse de un solo archivo almacenado de *registros jerárquicos* almacenados, consistentes cada uno en (a) el número, nombre y presupuesto de un departamento dado, junto con (b) el número, nombre y salario de cada empleado que pertenezca a ese departamento. En otras palabras, los datos pueden estar almacenados físicamente en cualquier forma que parezca adecuada, aunque en el nivel lógico siempre luzcan igual.

TRANSACCIONES

Nota: El tema de esta sección no es exclusivo de los sistemas relacionales. Sin embargo, lo cubrimos aquí debido a que es necesario entender la idea básica para poder apreciar ciertos aspectos del material que se encuentra en la parte II. No obstante, en forma deliberada, nuestra cobertura en este punto no es muy profunda.

En el capítulo 1 dijimos que una transacción es una *unidad de trabajo lógica* que comprende por lo regular varias operaciones de base de datos. También indicamos que el usuario debe ser capaz de informar al sistema cuando haya operaciones distintas que forman parte de la misma transacción. Para este fin, se proporcionan las operaciones BEGIN TRANSACTION, COMMIT y ROLLBACK (iniciar transacción, confirmar y deshacer). En esencia, una transacción comienza cuando se ejecuta una operación BEGIN TRANSACTION y termina cuando se ejecuta una operación COMMIT o ROLLBACK correspondiente. Por ejemplo (en seudocódigo):

```
                                                    /* retiro
                                                    /* depósito
BEGIN TRANSACTION ; /* Pasa dinero de la cuenta A a la cuenta S
UPDATE cuenta A ;
UPDATE cuenta 6 ;
IF todo salió bien
   THEN COMMIT ;                                    /* terminación normal  */
   ELSE ROLLBACK ; END IF ;                         /* terminación anormal */
```

Algunas ideas que surgen, son las siguientes:

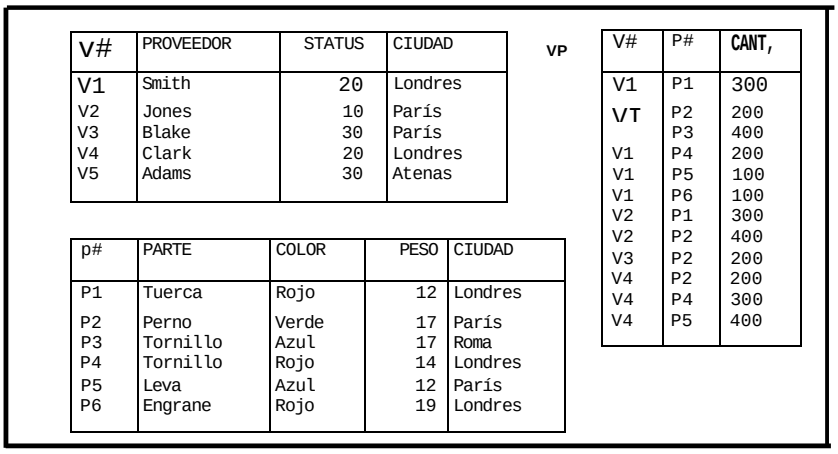
1. Se garantiza que las transacciones sean **atómicas;** es decir, se garantiza (desde un punto de vista lógico) que se ejecuten en su totalidad o que no se ejecuten en lo absoluto, aun cuando (por decir algo) el sistema fallara a la mitad del proceso.
2. También se garantiza que las transacciones sean **durables,** en el sentido de que una vez que una transacción ejecuta con éxito un COMMIT, debe garantizar que sus actualizaciones sean aplicadas a la base de datos, aun cuando el sistema falle en cualquier punto. *Nota:* Fundamentalmente, es esta propiedad de durabilidad de las transacciones la que hace *persistente* a la información de la base de datos en el sentido señalado en el capítulo i.
3. Se garantiza además que las transacciones estén **aisladas** entre sí, en el sentido de que las actualizaciones hechas a la base de datos por una determinada transacción *T1 no* sean visibles para ninguna transacción distinta *T2,* por lo menos hasta que *T1* ejecute con éxito el COMMIT. La operación COMMIT hace que las actualizaciones efectuadas pop una transacción sean visibles para otras transacciones; se dice que dichas actualizaciones están *confirmadas* y se garantiza que nunca sean canceladas. Si en vez de ello la transacción ejecuta un ROLLBACK, se cancelan *(deshacen)* todas las actualizaciones hechas por la transacción. En este último caso, el efecto es como si nunca se hubiese realizado la transacción.

4. Se garantiza (por lo regular) que la ejecución intercalada de un conjunto de transacciones concurrentes sea **seriable,** en el sentido que produzca el mismo resultado que se obtendría si se ejecutaran esas mismas transacciones, una a la vez, en un cierto orden serial no especificado.

Para una explicación más amplia de los puntos anteriores, además de muchos otros, consulte los capítulos 14 y 15.

3.9 LA BASE DE DATOS DE PROVEEDORES Y PARTES

Nuestro ejemplo a lo largo de gran parte del libro es la bien conocida base de datos de **provee dores y partes** (la cual mantiene, como vimos en el capítulo 1, una compañía ficticia de nom bre KnowWare Inc.). La finalidad de esta sección es explicar esa base de datos de manera que sirva como punto de referencia para capítulos posteriores. La figura 3.8 presenta un conjunto de valores de datos de ejemplo; de hecho, cuando haya alguna diferencia, los ejemplos subsi guientes tomarán estos valores específicos. La figura 3.9 muestra la definición de la base de datos, expresada una vez más en (una ligera variante de) **Tutorial D.** Observe en particular las especificaciones de las claves primaria y éxterna. Observe también que (a) varias columnas tienen tipos de datos con el mismo nombre que la columna en cuestión; (b) la columna STATUS y las dos columnas CIUDAD están definidas en términos de tipos integrados —INTEGER (en teros) y CHAR (cadenas de caracteres de longitud arbitraria)— en lugar de los tipos definidos por el usuario. Observe por último que hay un señalamiento importante con respecto a los valores de las columnas como se muestran en la figura 3.8, aunque aún no estamos en posición de hacer dicho señalamiento; regresaremos a él en el capítulo 5, sección 5.2, en la subsección titu lada "Representaciones posibles". \



V#

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

VP

V#	P#	CANT,
V1	P1	300
VT	P2	200
	P3	400
V1	P4	200
V1	P5	100
V1	P6	100
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200
V4	P4	300
V4	P5	400

p#	PARTE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	12	Londres
P2	Perno	Verde	17	París
P3	Tornillo	Azul	17	Roma
P4	Tornillo	Rojo	14	Londres
P5	Leva	Azul	12	París
P6	Engrane	Rojo	19	Londres

Figura 3.8 La base de datos de proveedores y partes (valores de ejemplo).

```
TYPE  V# ...
TYPE  NOMBRE      ;
TYPE  P# . . .
TYPE  COLOR . . .
TYPE  PESO ..
TYPE  CANT ..  ;

VAR  V BASE RELATION
     { V#        V#,
      PROVEEDOR NOMBRE,
      STATUS     INTEGER,
      CIUDAD     CHAR }
   PRIMARY KEY { V# } ;

VAR  P BASE RELATION
     { P#        P#,
      PARTE      NOMBRE,
      COLOR      COLOR,
      PESO       PESO,
      CIUDAD     CHAR }
     PRIMARY  KEY { P# } ;

VAR  VP BASE  RELATION
     { V#       V#,
       P#       P#,
       CANT     CANT }
     PRIMARY  KEY { V#, P# }
     FOREIGN  KEY { V# } REFERENCES V
     FOREIGN  KEY { P# } REFERENCES P ;
```

Figura 3.9 La base de datos de proveedores y partes (definición de datos).

La base de datos debería ser entendida como sigue:

La varrel V representa a los *proveedores.* Cada uno de ellos tiene un número de proveedor (V#), que es único para ese proveedor; un nombre de proveedor (PROVEEDOR), que no necesariamente es único (aunque en la figura 3.8 los valores de PROVEEDOR sí son únicos); un valor de clasificación o estado (STATUS); y una localidad (CIUDAD). Damos por hecho que cada proveedor está ubicado exactamente en una ciudad.

La varrel P representa las *partes* (con más precisión, las *clases* de partes). Cada clase de parte tiene un número de parte (P#), que es único; un nombre de parte (PARTE); un color (COLOR); un peso (PESO); y una ubicación en donde están almacenadas las partes de ese tipo (CIUDAD). Damos por hecho —donde haya alguna diferencia— que los pesos de las partes están dados en libras. También damos por hecho que cada clase de parte viene en un solo color y está almacenada en una bodega en una sola ciudad.

La varrel VP representa los *envíos.* En cierto modo, sirve para vincular las otras dos varrels, hablando de manera lógica. Por ejemplo, como muestra la figura 3.8, la primera fila de VP vincula un proveedor específico de V (es decir, el proveedor VI) con una parte específica de P (es decir, la parte Pl); en otras palabras, representa un envío de partes de la clase Pl por el proveedor V1 (y la cantidad del envío es de 300). Por lo tanto, cada envío tiene un número de proveedor (V#), un número de parte (P#) y una cantidad (CANT). Damos por

hecho que en un momento dado puede haber un envío como máximo para un proveedor y parte dados; por lo tanto, para un envío dado la combinación del valor V# y el valor P#es única con respecto al conjunto de envíos que aparecen actualmente en VP. *Nota:* Los valores de ejemplo de la figura 3.8 incluyen deliberadamente un proveedor, V5, que no tiene envío alguno correspondiente.

Subrayamos que (como señalé en el capítulo 1, sección 1.3) los proveedores y las partes pueden ser vistas como **entidades,** y un envío como un **vínculo** entre un proveedor en particular y una parte en particular. Sin embargo, como también señalé en esa sección, es mejor considerar un vínculo como un caso especial de una entidad. Una ventaja de las bases de datos relacionales es precisamente que todas las entidades, sin importar si son de hecho vínculos, están representadas de la misma manera uniforme; es decir, por medio de filas en relaciones, como indica el ejemplo.

Un par de observaciones finales:

- Primero, es claro que la base de datos de proveedores y partes es extremadamente sencilla, mucho más sencilla de lo que probablemente sea cualquier base de datos real; la mayoría de las bases de datos reales incluirán muchas más entidades y vínculos (y muchas más *cla ses* de entidades y vínculos) de los que tiene ésta. Sin embargo, ésta es adecuada al menos para ilustrar la mayoría de los puntos que necesitamos exponer en el resto del libro y (como ya señalé) la usaremos como base para la mayoría de —no todos— nuestros ejemplos en los capítulos siguientes.
- Segundo, es claro que no hay nada de malo en usar nombres más descriptivos como PROVEEDORES, PARTES y ENVÍOS en vez de los nombres más bien planos como V, P y VP que usamos arriba; de hecho, en la práctica se recomiendan generalmente los nom bres descriptivos. Pero en el caso específico de proveedores y partes, en los capítulos que siguen hacemos referencia tan a menudo a las varrels que nos pareció conveniente usar esos nombres cortos. Los nombres largos tienden a ser irritantes cuando se repiten mucho.

3.10 RESUMEN

Esto nos lleva al final de nuestro breve repaso de la tecnología relacional. Aunque la idea general del capítulo fue la de servir como una introducción concisa de explicaciones futuras mucho más extensas, es obvio que apenas hemos tocado la superficie de lo que en la actualidad se ha convertido en un tema muy amplio. Aún así, nos las arreglamos para cubrir bastante terreno. El siguiente es un resumen de los temas principales expuestos.

Una **base de datos relacional** es una base de datos que los usuarios perciben como un conjunto de **variables de relación** —es decir, *varrels*— o, de manera más informal, **tablas.** Un **sistema relacional** es aquel que maneja bases de datos y operaciones relacionales en dichas bases de datos, incluyendo las operaciones **restringir, proyectar** y **juntar** en particular. Todas estas operaciones, y otras parecidas, se encuentran en el **nivel de conjunto.** La propiedad de **cierre** de los sistemas relacionales significa que la salida de toda operación es la misma clase de objeto que la entrada (todas son relaciones), lo que significa que podemos escribir expresiones **relacionales anidadas.** Las varrels pueden actualizarse mediante la operación de **asignación** relacional: las operaciones **INSERT, UPDATE** y **DELETE** conocidas pueden ser vistas como abreviaturas de ciertas asignaciones relacionales comunes.

La teoría formal subyacente a los sistemas relacionales se denomina **modelo relacional de datos.** Este modelo se ocupa sólo de aspectos lógicos, no de aspectos físicos. Aborda Jos tres principales aspectos de los datos: la **estructura** de los datos, la **integridad** de los datos y la **manipulación** de los datos. El aspecto *estructural* tiene que ver con las relaciones *como tales;* el aspecto de *integridad* tiene que ver (entre otras cosas) con las **claves primaria y externa;** y el aspecto de *manipulación* tiene que ver con los operadores (restringir, proyectar, juntar, etcétera). ***El principio de información*** establece que todo el contenido de información de una base de datos relacional está representado en una y sólo una forma; es decir, como valores explícitos en posiciones de columnas dentro de filas en relaciones.

Cada relación tiene un **encabezado y** un **cuerpo;** el encabezado es un conjunto de parejas nombre-de-columna:nombre-de-tipo, mientras que el cuerpo es un conjunto de filas que se apegan al encabezado. El encabezado de una relación dada puede ser visto como un **predicado** y cada fila en el cuerpo denota una cierta **proposición verdadera,** la cual se obtiene sustituyendo ciertos valores de **argumento** del tipo apropiado por los *indicadores de posición* o **parámetros** del predicado. En otras palabras, los *tipos* son (conjuntos de) cosas de las cuales podemos hablar y las *relaciones* son (conjuntos de) cosas que podemos decir acerca de las cosas de las que podemos hablar. Juntos, los tipos y las relaciones son **necesarios y suficientes** para representar cualquier dato que queramos (esto es, al nivel lógico).

El **optimizador** es el componente del sistema que determina cómo implementar las petir ciones del usuario (las cuales se ocupan del *qué,* no del *cómo).* Por lo tanto, ya que los sistemas relacionales asumen la responsabilidad de navegar alrededor de la base de datos almacenada para localizar los datos deseados, en ocasiones se les describe como sistemas de **navegación automática.** La optimización y la navegación automática son prerrequisitos para la **independencia física de los datos.**

El catálogo es un conjunto de varrels del sistema que contienen **descriptores** de los diversos elementos que son de interés para el sistema (varrels base, vistas, índices, usuarios, etcétera). Los usuarios pueden consultar el catálogo exactamente del mismo modo en que consultan sus propios datos.

Las varrels originales de una base de datos dada se denominan **varrels base** y sus valores se llaman **relaciones base;** a una relación que se obtiene de dichas relaciones base mediante alguna expresión relacional, se le denomina relación **derivada** (en ocasiones, a las relaciones base y derivadas en su conjunto se les conoce como relaciones **expresables).** Una **vista** es una varrel cuyo valor en un momento dado es una relación derivada (en términos generales, se puede concebir como una **varrel derivada);** el valor de dicha varrel en un momento dado, es cualquiera que sea el resultado de evaluar la **expresión de definición de vista** asociada. Observe, por lo tanto, que las varrels base tienen una *existencia independiente* y que las vistas no la tienen; éstas dependen de las varrels base aplicables. (Otra forma de decir lo mismo es que las varrels base son **autónomas,** pero las vistas no lo son.) Los usuarios pueden operar sobre las vistas exactamente de la misma manera (por lo menos en teoría) como operan sobre las varrels base. El sistema implementa operaciones sobre vistas sustituyendo las referencias para el nombre de la vista por la expresión de definición de la misma, convirtiendo así la operación en una operación equivalente sobre las variables de relación base.

Una **transacción** es una *unidad de trabajo lógica* que involucra por lo regular a varias operaciones de base de datos. Una transacción inicia cuando se ejecuta **BEGIN TRANSACTION** y termina cuando se ejecuta **COMMIT** (terminación normal) o **ROLLBACK** (terminación anormal). Las transacciones son **atómicas, durables y aisladas** entre sí. Se garantiza (por lo regular) que la ejecución intercalada de un conjunto de transacciones concurrentes sea **seriable.**

Por último, el ejemplo básico para la mayor parte del libro es la **base de datos de proveedores** y **partes.** Si aún no lo ha hecho, vale la pena que le dedique un poco de tiempo para familiarizarse con ese ejemplo; es decir, por lo menos debe saber qué varrels tienen qué columnas y cuáles son las claves primaria y externa (¡no es tan importante saber exactamente cuáles son los valores de los datos de ejemplo!).

REÍ

EJERCICIOS

3.1 Defina los siguientes términos:

base de datos relacional	navegación automática
catálogo	operación en el nivel de conjunto
cierre	optimización
clave externa	predicado
clave primaria	proposición
confirmar	proyección
DBMS relacional	restricción
deshacer	varrel base
junta	varrel derivada
modelo relacional	vista

3.2 Haga un bosquejo del contenido de las varrels TABLA y COLUMNA del catálogo de la base de datos de proveedores y partes.

3.3 Como expliqué en la sección 3.6, el catálogo se describe a sí mismo; es decir, incluye entradas para las varrels del propio catálogo. Amplíe la figura 3.6 para que incluya las entradas necesarias para las varrels TABLA y COLUMNA.

3.4 La siguiente es una consulta sobre la base de datos de proveedores y partes. ¿Qué es lo que

```
RESULT := ( ( V JOIN VP ) WHERE P# = 'P2' ) { V#, CIUDAD } ;
```

Nota: De hecho hay un ligero problema con esta consulta, que tiene que ver con el tipo de dalos de la columna P#. Regresaremos a este punto en el capítulo 5, sección 5.2 (subsección "Conversión de tipos"). También se aplican observaciones similares al siguiente ejercicio.

3.5 Suponga que la expresión a la derecha de la asignación del ejercicio 3.4 se usa en una definición de vista:

```
CREATE VIEW U AS
     ( ( V JOIN VP ) WHERE P# = ' P2' ) { V#, CIUDAD } ;
```

Ahora considere esta consulta

```
RESULT := ( U WHERE CIUDAD - 'Londres' ) { V# } ;
```

¿Qué hace esta consulta? Muestre lo que haría el DBMS al procesar esta consulta.

3.6 ¿Qué entiende por los términos atomicidad, durabilidad, aislamiento y seriabilidad (de transacción)?

REFERENCIAS Y BIBLIOGRAFÍA

3.1 E. F. Codd: "Relational Database: A Practical Foundation for Productivity", *CACM 25,* No. 2 (febrero, 1982). Reeditado en Robert L. Ashenhurst (ed.), *ACM Turing Award Lectures: The First Twenty Years 1966-1985.* Reading, Mass.: Addison-Wesley *(ACM Press Anthology Series,* 1987).

Éste es el artículo que presentó Codd con motivo de la recepción del premio ACM Turing Award 1981 por su trabajo sobre el modelo relaciona]. En él expone el bien conocido problema de la *acumulación de aplicaciones.* Parafraseando: "La demanda de aplicaciones de cómputo está creciendo con rapidez —tanto que los departamentos de sistemas de información (cuya responsabilidad es proporcionar dichas aplicaciones) están cada vez más atrasados en satisfacer esa demanda". Hay dos formas complementarias de atacar este problema:

1. Proveer a los profesionales de la tecnología de información con nuevas herramientas para incrementar su productividad;

2. Permitir a los usuarios finales interactuar directamente con la base de datos, ignorando así por completo a dichos profesionales.

Ambos enfoques son necesarios y en este artículo Codd presenta evidencias para sugerir que las bases necesarias para ambos las ofrece la tecnología relacional.

3.2 C. J. Date: "Why Relational?", en *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Un intento para ofrecer un resumen conciso, aunque razonablemente amplio, de las principales ventajas de los sistemas relacionales. Vale la pena repetir aquí la siguiente observación del artículo: Entre las numerosas ventajas de "adoptar el modelo relacional" hay una en particular que no puede dejar de ser enfatizada, y es *la existencia de una base teórica sólida.* Citando el texto:

"...[el modelo] relacional es en verdad diferente. Es diferente porque no es *ad hoc.* Los sistemas anteriores fueron *ad hoc;* tal vez hayan ofrecido soluciones a ciertos problemas importantes del momento, pero estaban apoyados por ninguna base teórica sólida. En contraste, los sistemas relacionales sí están apoyados por esa base... lo que significa que son *sólidos como la roca."*

"Gracias a esta base sólida, los sistemas relacionales se comportan en formas bien definidas; y (tal vez sin darse cuenta de este hecho) los usuarios tienen en mente un modelo de comportamiento sencillo, uno que les permite predecir de manera confiable lo que hará el sistema en una situación dada. No hay (o no debe haber) sorpresas. El que sean predecibles significa que las interfaces de usuario son fáciles de entender, documentar, enseñar, utilizar y recordar."

3.3 C. J. Date y Hugh Darwen: *Foundation for Object/Relational Databases: The Third Manifesto.* Reading, Mass.: Addison-Wesley (1998). Vea también el artículo general introductorio *"The Third Manifesto:* Foundation for Object/Relational Databases", en C. J. Date, Hugh Darwen y David McGoveran, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).

The Third Manifesto es una propuesta detallada, formal y rigurosa de la futura dirección de las bases de datos y los DBMS. Puede verse como un anteproyecto del diseño de un DBMS y de la interfaz del lenguaje para dicho DBMS. Se basa en los conceptos centrales clásicos **tipo,** valor, **variable y operador.** Por ejemplo, podríamos tener un tipo INTEGER; el entero "3" podría ser un valor de ese tipo; N podría ser una variable de ese tipo, cuyo valor en todo momento da es un valor entero (es decir, un valor de ese tipo); y "+" podría ser un operador que se aplicí los valores enteros (es decir, a valores de ese tipo).

RESPUESTAS A EJERCICIOS SELECCIONADOS

3.3 La figura 3.10 muestra las entradas de las varrels TABLA y COLUMNA (sólo se omiten las en tradas de las varrels propias del usuario). Obviamente no es posible dar valores precisos de CONT-COL y CONTFILA.

3.4 La consulta recupera el número de proveedor y la ciudad de los proveedores que suministran la parte P2.

3.5 El significado de esta consulta es "Obtener los números de proveedores en Londres que su nistran la parte P2". El primer paso para procesar la consulta es sustituir el nombre U por la expre sión que define a U. Lo que nos da:

```
( ( ( ( V JOIN VP ) WHERE P# = 'P2' ) { V#, CIUDAD } )
                                  WHERE CIUDAD = 'Londres' ) { V# }
```

Esto se simplifica a:

```
( ( V WHERE CIUDAD = 'Londres' ) JOIN
                                    ( VP WHERE P# = 'P2' ) ) { V# }
```

```
( ( V WHERE CIUDAD
```

Para una explicación más amplia, vea los capítulos 9 y 17.

TABLA					
	NOMTABLA	CONTCOL	CONTFILA		
COLUMNA	TABLA COLUMNA	(3)			
	NOMTABLA	NOMCOL	.....		
	TABLA TABLA TABLA COLUMNA COLUMNA	NOMTABLA CONTCOL CONTFILA NOMTABLA NOMCOL			

Figura 3.10 Entradas del catálogo (en bosquejo) para TABLA y COLUMNA.

CAPITULO 4

Introducción a SQL

4.1 INTRODUCCIÓN

Como señalé en el capítulo 1, SQL es el lenguaje estándar para trabajar con bases de datos relaciOnales y es soportado prácticamente por todos los productos en el mercado. Originalmente, SQL fue desarrollado en IBM Research a principios de los años setenta (vea las referencias [4.8-4.9] y [4.28]); fue implementado por primera vez a gran escala en un prototipo de IBM llamado System R (vea las referencias [4.1-4.3] y [4.11-4.13]), y posteriormente en numerosos productos comerciales de IBM (vea la referencia [4.20]) y de muchos otros fabricantes. En este capítulo presentamos una introducción al lenguaje SQL; otros aspectos adicionales, que tienen que ver con temas como la integridad, la seguridad, etcétera, serán descritos en capítulos posteriores. Todas nuestras explicaciones están basadas (salvo que se indique lo contrario) en la versión actual del estándar conocido informalmente como **SQL/92,** también como *SQL-92* o simplemente como *SQL2* [4.22-4.23]; el nombre oficial es **Estándar Internacional del Lenguaje de Base de Datos SQL (1992).**

Nota: Debemos agregar de inmediato que está por concluir el trabajo sobre **SQL3,** la continuación propuesta al estándar actual; esperamos su ratificación a finales de 1999. Por lo tanto, para el momento en que aparezca publicado este libro, el estándar actual podría ser "SQL/99", en vez de SQL/92. Sin embargo, pensamos que sería demasiado confuso basar nuestras explicaciones en SQL3, ya que —obviamente— ningún producto lo maneja todavía; por ello decidimos exponer el SQL3 por separado en el apéndice B. En todo caso, también debemos señalar que actualmente ningún producto maneja el SQL/92 en su totalidad;* en vez de ello, los productos manejan generalmente lo que podría llamarse "un superconjunto de un subconjunto" de SQL/92 (es decir, la mayoría de ellos no soportan algunos aspectos del estándar y sin embargo en otros aspectos lo exceden). Por ejemplo, el producto DB2 de IBM no soporta actualmente todas las características de integridad del SQL/92, pero va más allá del SQL/92 en sus reglas relativas a la actualización de vistas.

Algunas observaciones preliminares:

- En un principio, SQL fue diseñado para ser específicamente un "sublenguaje de datos". Sin embargo, con la incorporación en 1996 de la característica de PSM *(Módulos Almacenados*

*De hecho, no sería posible que algún producto manejara el SQL/92 en su totalidad, ya que, como está especificado actualmente, el SQL/92 contiene numerosos huecos, errores e inconsistencias. Para una explicación detallada sobre este punto, consulte la referencia [4.19].

Persistentes) al estándar, SQL se convirtió, en términos computacionales, en un lenguaje completo (ahora incluye instrucciones como CALL, RETURN, SET, CASE. IF, LOOP, LEAVE, WHILE y REPEAT, así como diversas características relacionadas como las variables y los manejadores de excepciones). Como consecuencia, ya no hay necesidad de combinar SQL con algún lenguaje "anfitrión" distinto para desarrollar una aplicación completa. Sin embargo, en este libro decidimos no abordar los PSM en detalle. No le sorprenderá saber que SQL utiliza el término **tabla** en vez de los dos términos *relación* y *varrel* (vea el capítulo 3). Por lo tanto, para ser consistentes con el estándar y los productos SQL, haremos algo parecido en este capítulo (y en el resto del libro, siempre que nos ocupemos en forma específica de SQL). Además, SQL no emplea los términos *encabezado* y *cuerpo* (de una tabla o relación).

SQL es un lenguaje enorme. El documento estándar [4.22] supera por sí mismo las 600 páginas (y las especificaciones de SQL3 son de más del doble de ese tamaño). Como consecuencia, en un libro como éste no es posible abordar el tema en forma extensa; todo lo que podemos hacer es describir los principales aspectos de manera razonablemente amplia, aunque debo advertirle que nuestras explicaciones son superficiales por necesidad. En particular, no dudamos en omitir material irrelevante para los fines presentes ni en hacer simplificaciones en interés de la brevedad. Usted puede encontrar descripciones más completas (aunque aún en forma de tutoriales) en las referencias [4.4], [4.19] y [4.27],

Por último, debo decir que (como indiqué en varios puntos de los capítulos 1 al 3) SQL está muy lejos de ser el lenguaje relacional perfecto; padece de faltas tanto de omisión como de comisión. Sin embargo, es el estándar, es soportado por casi todos los productos del mercado y todo profesional de bases de datos debe saber algo acerca de él. De ahí su cobertura en este libro.

4.2 GENERALIDADES

SQL incluye operaciones tanto de definición como de manipulación de datos. Consideraremos primero las operaciones de definición. La figura 4.1 ofrece una definición de SQL para la base de datos de proveedores y partes (compárela y cotéjela con la figura 3.9 del capítulo 3). Como puede ver, la definición incluye una instrucción **CREATE TABLE** para cada una de las tres tablas base (como señalé en el capítulo 3, la palabra clave TABLE de CREATE TABLE significa específicamente tabla base). Cada una de estas instrucciones CREATE TABLE especifica el nombre de la tabla base que va a ser creada, los nombres y tipos de datos de las columnas de la tabla y las claves primaria y externa de esa tabla (posiblemente también alguna información adicional que no ilustramos en la figura 4.1). Surgen un par de cuestiones de sintaxis:

- Observe que a menudo usamos el carácter "#" (por ejemplo) en nombres de columnas, aunque de hecho ese carácter no es válido en SQL/92.
- Usamos el punto y coma ";" como terminador de una instrucción. El hecho que SQL/92 use en realidad dicho terminador depende del contexto. Los detalles exceden el alcance de este libro.

```
CREATE TABLE   V
     ( V#          CHAR(5),
       PROVEEDOR   CHAR(20),
       STATUS      NUMERIC(5),
       CIUDAD      CHAR(15),
     PRIMARY   KE  ( V# ) ) ;
CREATE TABLE   P
     ( P#          CHAR(6),
       PARTE       CHAR(20),
       COLOR       CHAR(6),
       PESO        NUMERIC(5,1),
       CIUDAD      CHAR(15),
     PRIMARY   KE  ( P# ) ) ;
CREATE TABLE   VP
     ( V#          CHAR(5),
       P#          CHAR(6),
       CANT        NUMERIC(9),
     PRIMARY   KE  ( V#, P# ),
     FOREIGN   KE  ( V# ) REFERENCES V,
     FOREIGN   KE  ( P# ) REFERENCES P ;
               Y
```

Figura 4.1 La base de datos de proveedores y partes (definición SQL).

Una diferencia importante entre la figura 4.1 y su contraparte (la figura 3.9) del capítulo 3, es que la figura 4.1 no incluye nada que corresponda a las definiciones de tipos (es decir, instrucciones TYPE) de la figura 3.9. Por supuesto, la razón es que SQL no permite que los usuarios definan sus propios tipos;* de ahí que las columnas deban ser definidas sólo en términos de tipos *integrados* (definidos por el sistema). SQL soporta los siguientes tipos integrados, que más o menos se explican por sí mismos:

```
CHARACTER [ VARYING J (n)      INTEGER        DATE
BIT [ VARYING ] (n)             SMALLINT       TIME
NUMERIC (p,q)                   FLOAT (p)      TIMESTAMP
DECIMAL (p,q)                                  INTERVAL
```

También soporta diversos valores predeterminados, abreviaturas y escrituras alternas —por ejemplo, CHAR por CHARACTER—; aquí omitimos los detalles. También, los corchetes "[" y "]" en CHARACTER y BIT significan que el material que encierran es opcional (como es normal, por supuesto, en la BNF [Forma Backus-Naur]). Por último, observe que SQL requiere que se especifiquen ciertas *longitudes* o *precisiones* para determinados tipos de datos (por ejemplo, CHAR), lo que no sucedía con nuestra sintaxis hipotética del capítulo 3. De hecho, aparentemente SQL considera esas longitudes y precisiones como parte del tipo (lo que implica que, por ejemplo, CHAR(3) y CHAR(4) sean tipos diferentes); pensamos que es mejor considerarlas como *restricciones de integridad* (y así lo hacemos; vea el capítulo 8, en especial el ejercicio 8.4).

Una vez definida la base de datos, podemos ahora comenzar a operar en ella por medio de las operaciones SQL de **manipulación** SELECT, INSERT, UPDATE y DELETE. En particular,

---

*SQL permite definir lo que llama *dominios*, pero dichos "dominios" no son realmente dominios —es decir, tipos— en el sentido relacional (vea el capítulo 5). *Nota:* Sin embargo, los tipos definidos por el usuario sí son soportados en SQL3 (vea el apéndice B).

podemos realizar operaciones relacionales de restringir, proyectar y juntar sobre los datos. ■ pleando en cada caso la instrucción de manipulación de datos **SELECT** de SQL. La figura 4.2 muestra cómo se pueden formular ciertas operaciones de restricción, proyección y junta en SQL. *Nota:* El ejemplo de junta de esa figura ilustra la idea de que en ocasiones son necesari los **nombres calificados** (por ejemplo, V. V#, VP. V#) para evitar la ambigüedad en las referencias a columnas. La regla general es que los nombres calificados siempre son aceptables, aunque los nombres no calificados también lo son en tanto no generen ambigüedad.

Subrayamos que SQL también soporta una forma abreviada de la cláusula SELECT, como lo ilustra el ejemplo siguiente:

```
SELECT *        — o "SELECT V.*    (es decir, el      puede ser calificado)
FROM V
```

El resultado es una copia de toda la tabla V; el asterisco es una abreviatura de una lista separa por comas con todos los nombres de columna de las tablas a las que se hace referencia en 1, cláusula FROM (de izquierda a derecha según el orden en el que están definidas dichas columñas en las tablas). Por cierto, observe el **comentario** en este ejemplo (presentado por un guión doble y terminado con un carácter de línea nueva). *Nota:* La expresión SELECT * FROM *T* donde *T* es un nombre de tabla) puede abreviarse aún más a TABLE *T*.

Explicamos la instrucción SELECT de manera mucho más amplia en el capítulo 7 (sección *1*

Restringir:

```
SELECT V#, P#, CANT
FROM VP
WHERE CANT < 150 ;
```

Resultado:

V#	p#	CANT
V1	P5	100
V1	P6	100

Proyectar:

```
SELECT V#, CIUDAD
FROM V ;
```

Resultado:

V#	CIUDAD
V1	Londres
V2	París
V3	París
V4	Londres
V5	Atenas

Juntar:

```
SELECT V.V#, PROVEEDOR, STATUS, CIUDAD, P#, CANT
FROM V, VP
WHERE V.V# = VP.V# ;
```

Resultado:

V#	PROVEEDOR	STATUS	CIUDAD	P#	CANT
V1	Smith	20	Londres	P1	300
V1	Smith	20	Londres	P2	200
V1	Smith	20	Londres	P3	400
V4	Clark	20	Londres	P5	400

Figura 4.2 Ejemplos de las operaciones restringir, proyectar y juntar en SQL.

Pasemos ahora a las operaciones de **actualización:** En el capítulo 1 dimos ejemplos de las instrucciones INSERT, UPDATE y DELETE de SQL, pero deliberadamente dichos ejemplos sólo comprendían operaciones de una fila. Sin embargo, al igual que SELECT, las operaciones **INSERT, UPDATE y DELETE** se encuentran en general en el *nivel de conjunto* (y de hecho, algunos ejercicios y respuestas del capítulo 1 ilustraron esta idea). Aquí tenemos algunos ejemplos de actualización en el nivel de conjunto para la base de datos de proveedores y partes:

```
INSERT
INTO   TEMP ( P#, PESO )
       SELECT P#, PESO
       FROM   P
       WHERE  COLOR = 'Rojo' ;
```

Este ejemplo da por hecho que ya creamos otra tabla TEMP con dos columnas, P# y PESO. La instrucción INSERT inserta en esa tabla los números de parte y los pesos correspondientes para todas las partes de color rojo.

```
UPDATE V
SET    STATUS • STATUS * 2
WHERE CIUDAD = 'París' ;
```

Esta instrucción UPDATE duplica el status de todos los proveedores que se encuentran en París.

```
DELETE
FROM  VP
WHERE P# = 'P2' ;
```

Esta instrucción DELETE elimina todos los envíos de la parte P2.

Nota: SQL no incluye un equivalente directo de la operación de **asignación** relaciona!. Sin embargo, podemos simular esa operación eliminando primero todas las filas de la tabla de destino y realizando después en esa tabla un INSERT... SELECT... (como en el primer ejemplo anterior).

4.3 EL CATALOGO

El estándar de SQL incluye especificaciones para un catálogo estándar denominado **Esquema de información.** De hecho, los términos convencionales "catálogo" y "esquema" son usados en SQL, aunque con un significado muy específico de ese lenguaje; hablando en términos generales, un **catálogo** de SQL consiste en los descriptores de una base de datos individual,* mientras que un **esquema** de SQL consiste en los descriptores de esa parte de la base de datos que pertenece a un usuario individual. En otras palabras, puede haber cualquier cantidad de catálogos (uno por base de datos), cada uno dividido en cualquier número de esquemas. Sin embargo, es necesario que cada catálogo incluya exactamente un esquema denominado INFORMATION_SCHEMA y, desde la perspectiva del usuario, es ese esquema (como ya indicamos) el que realiza la función normal de catálogo.

---

* En aras de la precisión, debemos mencionar que en el estándar de SQL ¡no existe el término "base de datos"! La forma exacta con que se denomina al conjunto de datos que describe el catálogo, está definida en la implementación. Sin embargo, es razonable pensar en dicho conjunto como en una base de datos.

El Esquema de información consiste entonces en un conjunto de tablas de SQL **cuyo** contenido en efecto refleja con precisión todas las definiciones de los demás esquemas del catálogo en cuestión. Para ser más exactos, el Esquema de información está definido para contener un conjunto de vistas de un "Esquema de definición" hipotético. No es necesario que la implementación soporte el Esquema de definición como tal, pero sí requiere que (a) soporte *algún* tipo de "Esquema de definición" **y** (b) soporte vistas de ese "Esquema de definición" que luzcan como las del Esquema de información. Los puntos que destacan son:

1. El razonamiento para establecer los requerimientos en términos de dos partes independientes (a) **y** (b) como acabamos de describir, es como sigue. Primero, los productos existentes soportan en efecto algo parecido al "Esquema de definición". Sin embargo, esos "Esquema de definición" varían ampliamente de un producto a otro (aun cuando los productos en cuestión sean del mismo fabricante). De ahí que tenga sentido la idea de requerir que la im plementación soporte solamente ciertas vistas predefinidas de su "Esquema de definición".
2. En realidad decimos "un" (no "el") Esquema de información ya que, como hemos visto, hay uno de estos esquemas en todo catálogo. Por lo tanto, la totalidad de los datos disponibles para un usuario dado generalmente *no* estarán descritos por un solo Esquema. de información. Sin embargo, para efectos de simplicidad, continuaremos hablando como si sólo fuera uno.

No vale la pena entrar en detalles sobre el contenido del Esquema de información; en su lugar, listaremos algunas de las vistas más importantes de este Esquema, con la esperanza de que sus nombres puedan por sí mismos dar una idea de lo que esas vistas contienen. Sin embargo debemos decir que la vista TABLES contiene información de *todas* las tablas nombradas (tanto de vistas como de tablas base), mientras que la vista VIEWS contiene información solamente de vistas.

```
SCHEMATA                        REFERENTIALCONSTRAINTS
DOMAINS                         CHECKCONSTRAINTS
TABLES                          KEY_COLUMN_USAGE
VIEWS                           ASSERTIONS
COLUMNS                         VIEW_TABLE_USAGE
TABLE_PRIVILEGES                VIEW_COLUMN_USAGE
COLUMN_PRIVILEGES               CONSTRAINT_TABLE_USAGE
USAGE_PRIVILEGES                CONSTRAINT_COLUMN_USAGE
DOMAIN_CONSTRAINTS              CONSTRAINT_DOMAIN_USAGE
TABLE_CONSTRAINTS
```

La referencia [4.19] ofrece más detalles; en particular, muestra cómo formular consultas frente al Esquema de información (el proceso no es tan directo como usted podría esperar).

4.4 VISTAS

El siguiente es un ejemplo de una definición de vista de SQL:

```
CREATE VIEW BUEN_PROVEEDOR
   AS SELECT V#, STATUS, CIUDAD
      FROM  V WHERE STATUS >
      15 ;
```

y éste es un ejemplo de una consulta SQL frente a esta vista:

```
SELECT V#, STATUS FROM
BUEN_PROVEEDOR WHERE CIUDAD
■ 'Londres' ;
```

Sustituyendo la definición de la vista por la referencia al nombre de la vista, obtenemos una expresión como la siguiente (observe la **subconsulta anidada** en la cláusula FROM):

```
SELECT BUEN_PROVEEDOR.V#,  BUEN_PROVEEDOR. STATUS
FROM ( SELECT V#, STATUS, CIUDAD
       FROM  V
       WHERE STATUS > 15 )  AS BUEN_PROVEEDOR
WHERE BUEN_PROVEEDOR.CIUDAD = 'Londres' ;
```

Y esta expresión puede simplificarse en algo como esto:

```
SELECT V#,  STATUS
FROM   V
WHERE  STATUS > 15
AND    CIUDAD = 'Londres'  ;
```

De hecho, esta última es la consulta que se ejecuta.

A manera de otro ejemplo, considere la siguiente operación DELETE:

```
DELETE
FROM  BUENPROVEEDOR
WHERE CIUDAD = 'Londres'
```

El DELETE que en realidad se ejecuta luce similar al siguiente:

```
DELETE
FROM  V
WHERE STATUS > 15
AND    CIUDAD =  'Londres'
```

4.5 TRANSACCIONES

En SQL, las instrucciones análogas de nuestro COMMIT y ROLLBACK se llaman **COMMIT WORK y ROLLBACK WORK,** respectivamente (en ambos casos, la palabra reservada WORK es opcional). Sin embargo, SQL no incluye explícitamente ninguna instrucción BEGIN TRANSACTION. En su lugar, cada que el programa ejecute una instrucción de **"inicio de transacción"** y no tenga una transacción en progreso, se iniciará implícitamente una transacción. Los detalles de cuáles instrucciones de SQL son de inicio de transacción estarían fuera de lugar en este texto; basta decir que todas las instrucciones expuestas en este capítulo son de inicio de transacción (por supuesto, con excepción de COMMIT y ROLLBACK).

4.6 SQL INCRUSTADO

La mayoría de los productos SQL permiten la ejecución de instrucciones SQL de **manera directa** (es decir, en forma interactiva desde una terminal en línea) y también como parte de un programa de aplicación (es decir, las instrucciones SQL pueden estar **incrustadas,** lo que significa que

pueden estar entremezcladas con las instrucciones del lenguaje de programación de dicho programa). Es más, en el caso de que las instrucciones estén incrustadas, el programa de aplicación puede estar escrito comúnmente en una variedad de lenguajes anfitrión (como COBOL, Java. PL/1, etcétera).* En esta sección consideraremos específicamente el caso de la incrustación.

El principio fundamental subyacente al SQL incrustado, al cual nos referiremos como el **principio de modo dual,** es que *toda instrucción SQL que puede ser usada en forma interactiva, también puede ser usada en un programa de aplicación.* Por supuesto, hay varias diferencias de detalle entre una determinada instrucción SQL interactiva y su contraparte incrustada —en especial, las instrucciones de recuperación requieren de un tratamiento más amplio en un entorno de programa anfitrión (vea más adelante en esta sección)—; sin embargo, el principio es muy cierto. (Cabe mencionar que su opuesto no lo es. Como veremos, muchas instrucciones del SQL incrustado no pueden ser usadas en forma interactiva.)

Antes de poder explicar las instrucciones reales del SQL incrustado, es necesario cubrir algunos detalles preliminares. La mayoría de estos detalles se ilustran en el fragmento de programa de la figura 4.3. (Para ordenar nuestras ideas, damos por hecho que el lenguaje anfitrión es PL/I. Gran parte de las ideas pueden traducirse a otros lenguajes anfitrión con cambios menores.) Los puntos que destacan son:

1. Las instrucciones del SQL incrustado están precedidas por EXEC **SQL,** para distinguirlas de las instrucciones del lenguaje anfitrión, y terminan con un símbolo **terminador** especial (un punto y coma en el caso de PL/I).
2. Una instrucción SQL *ejecutable* (en lo que resta de esta sección omitiremos en su mayoría el calificador "incrustado") puede aparecer en cualquier parte en donde aparezca una ins trucción ejecutable de lenguaje anfitrión. Por cierto, observe ese "ejecutable". A diferencia del SQL interactivo, el SQL incrustado incluye algunas instrucciones que son puramente declarativas, no ejecutables. Por ejemplo, DECLARE CURSOR no es una instrucción eje cutable (vea más adelante la subsección "Operaciones que involucran cursores"), tampoco

```
EXEC SQL BEGIN  DECLARE SECTION ;

    DCL SQLSTATE CHAR(5) ;

    DCL P#       CHAR(6) ;
    DCL PESO     FIXED DECIMAL(5  ,1) ;

EXEC SQL END DECLARE SECTION ;

P# = 'P2'                     /*  por ejemplo                          */

EXEC SQL SELECT P.PESO
         INTO    :PESO
         FROM    P
         WHERE   P.P# = :P# ;
IF SQLSTATE •   00000'
THEN ... ;                    /•  PESO = valor recuperado              */
ELSE ... ;                    /*  ocurrió alguna excepción             */
```

Figura 4.3 Fragmento de un programa PL/I con SQL incrustado.

*E1 estándar de SQL [4.22] soporta actualmente Ada, C, COBOL, Fortran, M (antes llamado MUMPS), Pascal y PL/I. Al momento de la publicación de este libro no estaba incluido el soporte de Java, pero debe ser incorporado pronto (vea la referencia [4.6] y también el apéndice B); algunos productos ya lo soportan.

son ejecutables BEGIN y END DECLARE SECTION (vea más adelante el párrafo 5), ni WHENEVER (vea el párrafo 9).

3. Las instrucciones de SQL pueden incluir referencias a **variables anfitrión;** estas referencias deben incluir un **prefijo de dos puntos** para distinguirlas de los nombres de columnas de SQL. Las variables anfitrión pueden aparecer en SQL incrustado en cualquier lugar donde aparezca una literal en SQL interactivo. También pueden aparecer en una cláusula INTO de SELECT (vea el párrafo 4) o de FETCH (una vez más, vea la subsección "Operaciones que involucran cursores") para designar destinos de operaciones de recuperación.
4. Observe la cláusula **INTO** de la instrucción SELECT de la figura 4.3. La finalidad de esta cláusula es (como acabamos de indicar) especificar las variables de destino en las que se recuperarán valores; la iésima variable de destino mencionada en la cláusula INTO corresponde al iésimo valor a recuperar, tal como se especifica en la cláusula SELECT.
5. Todas las variables anfitrión a las que se hace referencia en instrucciones SQL deben estar declaradas (DCL en PL/I) dentro de una **sección de declaración de SQL incrustado,** la cual está delimitada por las instrucciones **BEGIN y END DECLARE SECTION.**
6. Todo programa que contenga instrucciones de SQL incrustado debe incluir una variable anfitrión denominada **SQLSTATE.** Después ejecutar cualquier instrucción de SQL, un código de estado es devuelto al programa en dicha variable; en particular, un código de estado de 00000 significa que la instrucción se ejecutó con éxito, y un valor de 02000 significa que la instrucción se ejecutó pero no se encontraron datos para satisfacer la petición. Por lo tanto, toda instrucción SQL del programa debe en principio estar seguida de una comprobación de SQLSTATE y debe tomarse la acción adecuada si el valor no fue el esperado. Sin embargo, en la práctica dichas comprobaciones pueden estar implícitas (vea el párrafo 9).
7. Las variables anfitrión deben tener un **tipo de datos** apropiado de acuerdo con los usos para los que son puestas. En particular, una variable anfitrión que vaya a ser usada como un destino (por ejemplo, en SELECT) debe tener un tipo de datos que sea compatible con el de la expresión que proporciona el valor a asignar para ese destino; en forma similar, una variable anfitrión que vaya a ser utilizada como un origen (por ejemplo, en INSERT) debe tener un tipo de datos que sea compatible con el de la columna de SQL a la que se van a asignar los valores del origen. Se aplican observaciones similares a una variable anfitrión que va a ser empleada en una comparación o en cualquier otro tipo de operación. Consulte la referencia [4.22] para conocer los detalles de lo que significa exactamente que los tipos de datos sean compatibles en el sentido señalado.
8. Las variables anfitrión y las columnas de SQL pueden tener el mismo nombre.
9. Como ya mencioné, toda instrucción de SQL debe en principio estar seguida de una comprobación del valor que SQLSTATE devuelve. Para simplificar este proceso se incluye la instrucción **WHENEVER.** Esta instrucción toma la forma

```
EXEC SQL WHENEVER <condición> <acción>  ;
```

Aquí, <*condición*> puede ser SQLERROR o bien NOT FOUND y <*acción*> puede ser CONTINUE o bien una instrucción GO TO. WHENEVER no es una instrucción ejecutable; más bien es una directiva para el compilador de SQL. "WHENEVER <*condición*> GO TO <*etiqueta*>" hace que el compilador inserte una instrucción de la forma "IF <*condición*> GO TO <*etiqueta*> END IF" después de encontrar cada instrucción SQL ejecutable;

"WHENEVER <*condición*> CONTINUE" hace que no inserte ninguna de dichas instrucciones, lo que implica que el programador insertará a mano esas instrucciones. Las dos <*condición*> están definidas como sigue:

NOTFOUND	*significa*	no se encontraron datos; por lo regular, SQLSTATE = 02000
SQLERROR	*significa*	ocurrió un error; vea SQLSTATE en la referencia [4.22]

Cada instrucción WHENEVER encontrada por el compilador de SQL en su examen secuencial a través del texto del programa (en busca de una condición en particular), anula a la instrucción anterior encontrada (para esa condición).

10. Por último, observe que —para utilizar la terminología del capítulo 2— el SQL incrustado constituye un *acoplamiento débil* entre SQL y el lenguaje anfitrión.

Suficiente en lo que respecta a los preliminares. En el resto de esta sección nos concentraremos de manera específica en las operaciones de manipulación de datos. Como ya indiqué, la mayoría de esas operaciones pueden ser manejadas de manera bastante directa (es decir, con cambios menores en la sintaxis). Sin embargo, las operaciones de recuperación requieren de un tratamiento especial. El problema es que dichas operaciones recuperan (en general) muchas filas, no sólo una, y los lenguajes anfitrión por lo regular no están equipados para manejar la recuperación de más de una fila a la vez. Por lo tanto, es necesario proporcionar algún tipo de puente entre la capacidad de recuperación en el nivel de conjunto de SQL y la capacidad de recuperación en el nivel de fila del lenguaje anfitrión; dicho puente lo proporcionan los cursores. Un cursor es un tipo especial de objeto SQL que sólo se aplica al SQL incrustado (debido a que el SQL interactivo no lo necesita). En esencia, consiste en una clase de *apuntador {lógico)* que puede ser empleado para recorrer un conjunto de filas, apuntando en su momento a cada una de ellas y proporcionando por lo tanto una direccionabilidad para cada una de esas filas a la vez. Sin embargo, postergaremos la explicación detallada de los cursores para una subsección posterior y consideraremos primero aquellas instrucciones que no necesitan de cursores.

Operaciones que no involucran cursores

Las instrucciones de manipulación de datos que no necesitan cursores son las siguientes:

- SELECT individual
- INSERT
- UPDATE (excepto la forma CURRENT. Vea más adelante)
- DELETE (de nuevo, con excepción de la forma CURRENT. Vea más adelante)

A continuación damos ejemplos de cada una de estas instrucciones:

SELECT individual: Obtener el status y la ciudad de los proveedores cuyo número de proveedor está dado por la variable anfitrión V#DADO.

```
EXEC SQL SELECT STATUS, CIUDAD
         INTO  : CATEGORÍA, : CIUDAD
         FROM  V
         WHERE V# - :V#DAD0 ;
```

Empleamos el término *SELECT individual* para referirnos a una instrucción SELECT que produce una tabla que contiene una fila como máximo. En el ejemplo, si en la tabla V hay exactamente una fila que satisface la condición de la cláusula WHERE, entonces los valores de STATUS y CIUDAD de esa fila serán asignados a las variables CATEGORÍA y CIUDAD de acuerdo con lo solicitado y SQLSTATE quedará establecido como 00000; si ninguna fila de V satisface la condición WHERE, SQLSTATE quedará establecido como 02000; y si más de una fila satisface la condición, el programa está mal y se asignará un código de error a SQLSTATE.

INSERT: Insertar en la tabla P una parte nueva (número, nombre y peso de parte dados por las variables anfitrión P#, PARTE y PESOP, respectivamente; el color y la ciudad se desconocen).

```
EXEC SQL INSERT
         INTO  P ( P#, PARTE, PESO )
         VALUES ( :P#, :PARTE, : PESOP ) ;
```

A los valores COLOR y CIUDAD de la nueva parte se les asignarán los **valores predeterminados** aplicables. Para una explicación de los valores predeterminados de SQL, consulte el capítulo 5, sección 5.5.

UPDATE: Aumentar el status de todos los proveedores de Londres en una cantidad dada por la variable anfitrión AUMENTO.

```
EXEC SQL UPDATE V
         SET   STATUS = STATUS + ¡AUMENTO
         WHERE CIUDAD ■ 'Londres' ;
```

Si ninguna de las filas de proveedores satisface la condición WHERE, se asignará el valor 02000 a SQLSTATE.

DELETE: Eliminar todos los envíos de proveedores cuya ciudad esté dada por la variable anfitrión CIUDAD.

```
EXEC SOL DELETE
         FROM  VP
         WHERE  :CIUDAD =
              ( SELECT  CIUDAD FROM   V
                WHERE  V.V# = VP.V# ) ;
```

Una vez más, si ninguna fila de VP satisface la condición WHERE, se le asignará 02000 a SQLSTATE. Observe de nuevo la **subconsulta anidada** (esta vez, en la cláusula WHERE).

Operaciones que involucran cursores

Pasemos ahora a la cuestión de la recuperación en el nivel de conjunto; es decir, la recuperación de un conjunto que contiene un número arbitrario de filas, en lugar de una como máximo como en el caso del SELECT individual. Como expliqué antes, lo que se necesita aquí es un mecanismo para acceder a cada una de las filas del conjunto (una a la vez) y los **cursores** ofrecen este mecanismo. El proceso lo ilustra de manera esquemática el ejemplo de la figura 4.4, que está diseñado para recuperar los detalles (V#, PROVEEDOR y STATUS) de todos aquellos proveedores de la ciudad dada por la variable anfitrión Y.

Explicación: La instrucción "DECLARE X CURSOR ..." define un cursor de nombre X, con una *expresión de tabla* asociada (es decir, una expresión que da como resultado una tabla), especificada por el SELECT que forma parte de ese DECLARE. Dicha expresión de tabla no se

```
EXEC SQL DECLARE X CURSOR FOR     /* define el , cursor                */
     L   SELECT V.V#,             V.STATUS
         V.PROVEEDOR FROM  V
         WHERE V.CIUDAD = :Y
         ORDER  BY V# ASC ;

EXEC SQ  OPEN X ; DO para  filas  /* ejecuta la de a consulta nediante X  */
     L   todas las EXEC    X     V accesibles     ; :STATUS ; el         */
         SQL FETCH         INTO  :V#, ¡PROVEEDOR  siguiente proveedor
                                 /* recupera

EXEC SQ  END ;                      /  desactiva  el cursor X           */
     L   CLOSE X ;                  *
```

Figura 4.4 Ejemplo de recuperación de múltiples filas.

evalúa en este punto, DECLARE CURSOR es una expresión meramente declarativa. La expresión es producida al abrir el cursor ("OPEN X"). Entonces se usa la instrucción "FETCH X INTO ..." para recuperar (una a una) las filas del conjunto resultante asignando a las variables anfitrión los valores recuperados de acuerdo con las especificaciones de la cláusula INTO de esa instrucción. (Para efectos de simplicidad dimos a las variables anfitrión los mismos nombres de las columnas correspondientes de la base de datos. Observe que el SELECT en la declaración del cursor no tiene ninguna cláusula INTO propia.) Debido a que habrá muchas filas en el conjunto resultante, el FETCH aparecerá normalmente dentro de un ciclo; el ciclo se repetirá en tanto queden filas en ese conjunto. Al salir del ciclo, el cursor X se cierra ("CLOSE X").

Ahora consideraremos con más detalle los cursores y las operaciones con cursores. En primer lugar, un cursor se declara por medio de una instrucción **DECLARE CURSOR,** la cual tiene la siguiente forma general

```
EXEC SQL DECLARE <nombre de cursor> CURSOR
         FOR <expresión de tabla> [ <ordenamiento> ] ;
```

(para ser breves, ignoramos algunas especificaciones opcionales). Para una explicación completa de <*expresión de tabla* >, consulte el apéndice A. El <*ordenamiento*> opcional toma la form

```
ORDER BY <lista de elementos a ordenar separados con comas>
```

donde (a) la lista separada con comas de <*elemento a ordenar*> no debe estar vacía —vea el párrafo inmediato siguiente— y (b) cada <*elemento a ordenar*> individual consiste en un nombre de columna *{no* calificado, cabe señalar), seguido de manera opcional por ASC (ascendente) o DESC (descendente), con ASC como el valor predeterminado.

Nota: Definimos el término *lista con comas* como sigue. Sea **<*xyz*>** un elemento para notar una categoría sintáctica arbitraria (es decir, todo lo que aparezca a la izquierda de una regla de producción BNF). Entonces la expresión < *lista de xyz separadas con comas*> denota una cuencia de cero o más elementos **<*xyz*>** en la cual cada par de <*xyz*> adyacentes está separado por una coma (y tal vez por uno o más espacios). Observe que haremos un uso amplio de esta notación en las reglas de sintaxis futuras (en todas las reglas, no sólo en las de SQL).

Como mencioné antes, la instrucción DECLARE CURSOR es declarativa, no ejecutable; declara un cursor con el nombre especificado y con la expresión de tabla y ordenamiento

pecificados en asociación permanente con él. La expresión de tabla puede incluir referencias a variables anfitrión. Un programa puede tener cualquier cantidad de instrucciones DECLARE CURSOR, cada una de las cuales debe ser (por supuesto) para un cursor diferente.

Para operar sobre los cursores se ofrecen las instrucciones ejecutables: **OPEN, FETCH** y **CLOSE.**

- La instrucción

```
EXEC SQL OPEN <nombre de cursor> ;
```

  abre o *activa* el cursor especificado (que no debe estar abierto actualmente). En efecto, la expresión de tabla asociada con el cursor es evaluada (empleando los valores actuales para cualquier variable anfitrión referida dentro de esa expresión); entonces se identifica un conjunto de filas y se convierte en el **conjunto activo** del cursor. El cursor también identifica una *posición* dentro de ese conjunto activo que es la posición inmediata anterior a la primera fila. (Para que el concepto de posición tenga sentido, se considera que los conjuntos activos siempre tienen un orden.* El orden puede ser el definido por la cláusula ORDER BY o bien por un ordenamiento determinado por el sistema en ausencia de dicha cláusula).

- La instrucción

```
EXEC SQL FETCH <nombre de cursor>
         INTO <lista de referencias a variable anfitrión separadas con comas> ;
```

  avanza el cursor especificado (que debe estar abierto) hacia la siguiente fila del conjunto activo y luego asigna el iésimo valor de esa fila a la iésima variable anfitrión referida en la cláusula INTO. Si no hay una siguiente fila al ejecutar FETCH, se asigna el valor 02000 a SQLSTATE y no se recuperan datos.

- La instrucción

```
EXEC SQL CLOSE <nombre de cursor> ;
```

  cierra o *desactiva* el cursor especificado (que debe estar abierto actualmente). Ahora el cursor no tiene ningún conjunto activo. Sin embargo, puede volverse a abrir después, en cuyo caso adquirirá otro conjunto activo; probablemente no el mismo que antes, en especial si en el transcurso cambió el valor de cualquiera de las variables anfitrión referidas en la declaración del cursor. Observe que modificar el valor de dichas variables mientras el cursor está abierto no tiene ningún efecto en el conjunto activo actual.

Otras dos instrucciones pueden incluir referencias a cursores; a saber, las formas **CURRENT** de **UPDATE y DELETE.** Si un cursor, digamos X, está ubicado actualmente en una fila en par-

---

* Por supuesto, los conjuntos por sí mismos no tienen un orden (vea el capítulo 5), de modo que un "conjunto activo" en realidad no es un conjunto como tal. Sería mejor considerarlo como una *lista ordenada* o *arreglo* (de filas).

ticular, entonces es posible modificar (UPDATE) o eliminar (DELETE) "la fila actual (CURRENT) de X"; es decir, la fila en la que está posicionado X. Por ejemplo:

```
EXEC SQL UPDATE V
         SET   STATUS • STATUS + :AUMENTO
         WHERE CURRENT OF X ;
```

Nota: Las formas CURRENT de UPDATE y DELETE no están permitidas cuando la expresión de tabla en la declaración del cursor define una vista no actualizable como parte de una instrucción CREATE VIEW (vea el capítulo 9, sección 9.6).

SQL dinámico

El **SQL dinámico** consiste de una serie de propiedades del SQL incrustado que están destinadas a apoyar la construcción de aplicaciones generales, en línea y posiblemente interactivas. (Recuerde del capítulo 1 que una aplicación en línea es aquella que maneja el acceso a la base de datos desde una terminal en línea). Considere lo que una aplicación en línea típica tiene que hacer. A grandes rasgos, los pasos que debe seguir son, los siguientes:

1. Aceptar un comando de la terminal.
2. Analizar ese comando.
3. Ejecutar las instrucciones SQL correspondientes en la base de datos.
4. Devolver a la terminal un mensaje o los resultados.

Si el conjunto de comandos que puede aceptar el programa en el paso 1 es relativamente pequeño, como es el caso (tal vez) de un programa que maneja reservaciones de una línea aérea entonces es probable que el conjunto de posibles instrucciones SQL a ejecutar también sea reducido y se pueda "alambrar en forma permanente" dentro del programa. En este caso, los pasos 2 y 3 anteriores consistirán simplemente en la lógica para examinar el comando de entrada y después segmentar a la parte del programa que emite las instrucciones SQL predefinidas. Por otra parte, si existe la posibilidad de una gran variación en la entrada, entonces podría no ser prác tico predefinir y "alambrar en forma permanente" las instrucciones SQL para todo comanado posible. En su lugar, probablemente sea más conveniente construir las instrucciones SQL nece-sarias de manera dinámica, y después compilar y ejecutar dichas instrucciones. Las propiedades del SQL dinámico ayudan en este proceso.

Las dos instrucciones dinámicas principales son PREPARE y EXECUTE. Su uso se ilustra en el siguiente ejemplo (de simplicidad irreal, pero preciso).

```
DCL SQLFUENTE CHAR VARYING (65000) ¡

SQLFUENTE ■ 'DELETE FROM VP WHERE CANT < 300' ;
EXEC SQL PREPARE SQLPREPARADO FROM : SQLFUENTE ;
EXEC SQL EXECUTE SQLPREPARADO ;
```

Explicación:

1. El nombre SQLFUENTE identifica una variable de PL/I de cadena de caracteres con longitud variable, en la que el programa construirá de alguna manera el código fuente (es decir, la representación en cadena de caracteres) de alguna instrucción SQL; en nuestro ejemp particular, una instrucción DELETE.

2. En contraste, el nombre SQLPREPARADO, identifica una variable de *SQL* —no una varia ble de PL/I— que (de manera conceptual) se usará para contener la forma compilada de las instrucciones SQL cuya forma fuente está dada en SQLFUENTE. (Por supuesto, los nom bres SQLFUENTE y SQLPREPARADO son arbitrarios).
3. La instrucción PL/I de asignación "SQLFUENTE = ... ;" asigna a SQLFUENTE la forma fuente de una instrucción DELETE de SQL. Desde luego, es probable que en la práctica el proceso de construir dicha instrucción fuente sea mucho más complejo (tal vez incluya la entrada y el análisis de alguna petición del usuario final, expresada en lenguaje natural o en alguna otra forma más fácil de usar que SQL).
4. La instrucción PREPARE toma entonces la instrucción fuente y la "prepara" (es decir, la compila) para producir una versión ejecutable, y que se almacena en SQLPREPARADO.
5. Por último, la instrucción EXECUTE ejecuta la versión SQLPREPARADO y ocasiona así que ocurra la eliminación real (DELETE). La información de SQLSTATE del DELETE se devuelve exactamente como si el DELETE se hubiera ejecutado directamente en la forma normal.

Observe que debido a que el nombre SQLPREPARADO denota una variable de SQL y no una variable de PL/I, éste no tiene el prefijo de dos puntos cuando se hace referencia a él en las instrucciones PREPARE y EXECUTE. Observe además que dichas variables de SQL no tienen que ser declaradas en forma explícita.

Por cierto, el proceso que acabamos de describir es exactamente lo que sucede cuando las propias instrucciones SQL son introducidas de manera interactiva. La mayoría de los sistemas ofrecen algún tipo de procesador de consultas SQL interactivo. De hecho, dicho procesador es sólo una clase particular de aplicación en línea general y está listo para aceptar una variedad muy amplia de entradas; es decir, cualquier instrucción SQL válida (o ¡inválida!). Utiliza las propiedades del SQL dinámico para construir instrucciones SQL adecuadas que correspondan a su entrada, para compilar y ejecutar esas instrucciones construidas y para devolver mensajes y resultados a la terminal.

Concluimos esta subsección (y la sección) con un breve comentario sobre una adición reciente (1995) al estándar conocido como la **Interfaz a nivel de llamada de SQL** ("SQL/CLI", *CLI* para abreviar). La interfaz CLI se basa en gran medida en la interfaz ODBC *(Conectividad Abierta de Base de Datos)* de Microsoft. La interfaz CLI permite que una aplicación escrita en uno de los lenguajes anfitrión usuales emita peticiones de base de datos llamando a ciertas *rutinas CLI* proporcionadas por el fabricante. Dichas rutinas, que deben haberse enlazado a la aplicación en cuestión, emplean entonces el SQL dinámico para realizar las operaciones de base de datos solicitadas, en beneficio de la aplicación. (En otras palabras, y desde el punto de vista del DBMS, las rutinas CLI pueden ser concebidas simplemente como otra aplicación.)

Como puede ver, SQL/CLI (y también ODBC) abordan el mismo problema general que el SQL dinámico: Ambos permiten escribir programas para los que no se conocen, hasta el momento de la ejecución, las instrucciones SQL exactas a ejecutar. Sin embargo, representan de hecho un mejor enfoque al problema que en el caso del SQL dinámico. Hay dos razones principales de esto:

- Primero, el SQL dinámico es un *código fuente* estándar. Por lo tanto, toda aplicación que use SQL dinámico requiere los servicios de algún tipo de compilador SQL a fin de procesar las operaciones —PREPARE, EXECUTE, etcétera— prescritas por ese estándar. En contraste, CLI simplemente estandariza los detalles de ciertas *llamadas a rutinas* (es decir,

llamadas a subrutinas); no se necesitan los servicios especiales de un compilador, sólo los servicios normales del compilador del lenguaje anfitrión estándar. Como resultado, las aplicaciones pueden distribuirse (tal vez a través de otros fabricantes de software) en forma de *código objeto* "comprimido".

- Segundo, dichas aplicaciones pueden ser *independientes del DBMS;* es decir, CLI incluye características que permiten la creación (una vez más, por parte de otros fabricantes de software) de aplicaciones genéricas que pueden ser usadas con varios DBMS distintos, en va de que tengan que ser específicas para un DBMS en particular.

En la práctica, las interfaces como CLI, ODBC y JDBC (una variante de Java para ODBC) se están haciendo cada vez más importantes, por razones que se exponen (en parte) en el capítulo 20.

4.7 SQL NO ES PERFECTO

Como mencioné en la introducción a este capítulo, SQL está muy lejos de ser el lenguaje relacional "perfecto"; padece de muchas faltas tanto de omisión como de comisión. En los capítulos subsiguientes ofreceremos críticas específicas en el punto apropiado, pero el aspecto sobresaliente es simplemente que SQL tiene muchas fallas en el manejo adecuado del modelo relacional. Como consecuencia, no está del todo claro que los productos SQL actuales merezcan en realidad la etiqueta de "relacionales". De hecho, hasta donde yo sé, *en el mercado actual no hay producto alguno que soporte todos los detalles del modelo relacional.* Esto no quiere decir que algunas partes del modelo no sean importantes; al contrario, *todos los detalles* lo son. Lo que es más, son importantes por razones auténticamente prácticas. En realidad, no podemos dejar de subrayar el hecho de que la finalidad de la teoría relacional no es sólo "una teoría para su propio beneficio"; más bien. la finalidad es ofrecer una base sobre la cual construir sistemas que sean *100 por ciento prácticos.* Pero la triste realidad es que los fabricantes aún no han enfrentado el reto de implementar la teoría en su totalidad. Como consecuencia, los productos "relacionales" de hoy no logran (de una u otra manera) cumplir por completo la promesa de la tecnología relacional.

4.8 RESUMEN

Con esto termina nuestra introducción a algunas de las principales características del estándar de SQL ("SQL/92"). Enfatizamos el hecho de que SQL es muy importante desde una perspectiva comercial; aunque por desgracia, deficiente en cierto modo desde un punto de vista puramente relacional.

SQL incluye tanto un componente de lenguaje de definición de **datos** (DDL) como un componente de lenguaje de **manipulación de datos** (DML). El DML de SQL puede operar tanto en el nivel externo (sobre vistas) como en el nivel conceptual (sobre tablas base). En forma similar, el DDL puede usarse para definir objetos en el nivel externo (vistas), en el nivel concepto (tablas base) e incluso —en la mayoría de los sistemas comerciales, aunque no en el estándí *como tal*— también en el nivel interno (es decir, índices y otras estructuras de almacenamiento físico). Además, SQL ofrece también ciertas propiedades de *control de datos;* es decir, propiedades que no pueden ser realmente clasificadas como pertenecientes a DDL o DML. Un ejemplo de estas propiedades es la instrucción GRANT, la cual permite a los usuarios conceder *privilegios de acceso* entre sí (vea el capítulo 16).

Mostramos la forma en que SQL puede ser usado para crear tablas base mediante la instrucción **CREATE TABLE.** Después dimos algunos ejemplos de las instrucciones **SELECT, INSERT, UPDATE** y **DELETE;** mostramos en particular cómo puede emplearse SELECT para expresar las operaciones relacionales de restringir, proyectar y juntar. También describimos brevemente el **Esquema de información,** el cual consiste en un conjunto de vistas prescritas de un "Esquema de definición" hipotético y vimos las propiedades de SQL para el manejo de **vistas** y **transacciones.**

Gran parte de este capítulo se ocupó del **SQL incrustado.** Aquí, la idea básica del SQL incrustado es el **principio del modo dual;** es decir, el principio de que (en la medida de lo posible) *toda instrucción SQL que puede ser usada en forma interactiva también puede ser usada en un programa de aplicación.* La principal excepción a este principio surge en conexión con las **operaciones de recuperación de múltiples filas,** la cual requiere de un **cursor** para cerrar la brecha entre las posibilidades de recuperación de SQL en el nivel de conjunto y las posibilidades de recuperación en el nivel de fila de los lenguajes anfitrión como PL/I.

Para seguir varios preparativos necesarios (aunque en su mayoría, de sintaxis) —incluida en particular una breve explicación de **SQLSTATE**— consideramos aquellas operaciones que no necesitan de cursores; es decir, **SELECT individual, INSERT, UPDATE** y **DELETE.** Después pasamos a las operaciones que *sí* requieren de cursores y explicamos **DECLARE CURSOR, OPEN, FETCH, CLOSE** y las formas **CURRENT** de **UPDATE** y **DELETE.** (El estándar hace referencia a las formas CURRENT de estos operadores como UPDATE y DELETE *posicionados,* y utiliza el término UPDATE y DELETE *examinados* para las formas que no son CURRENT). Por último, dimos una muy breve introducción al concepto de **SQL dinámico,** y en particular describimos las instrucciones **PREPARE** y **EXECUTE** y mencionamos también la **Interfaz en el nivel de llamada de SQL (CLI).**

EJERCICIOS

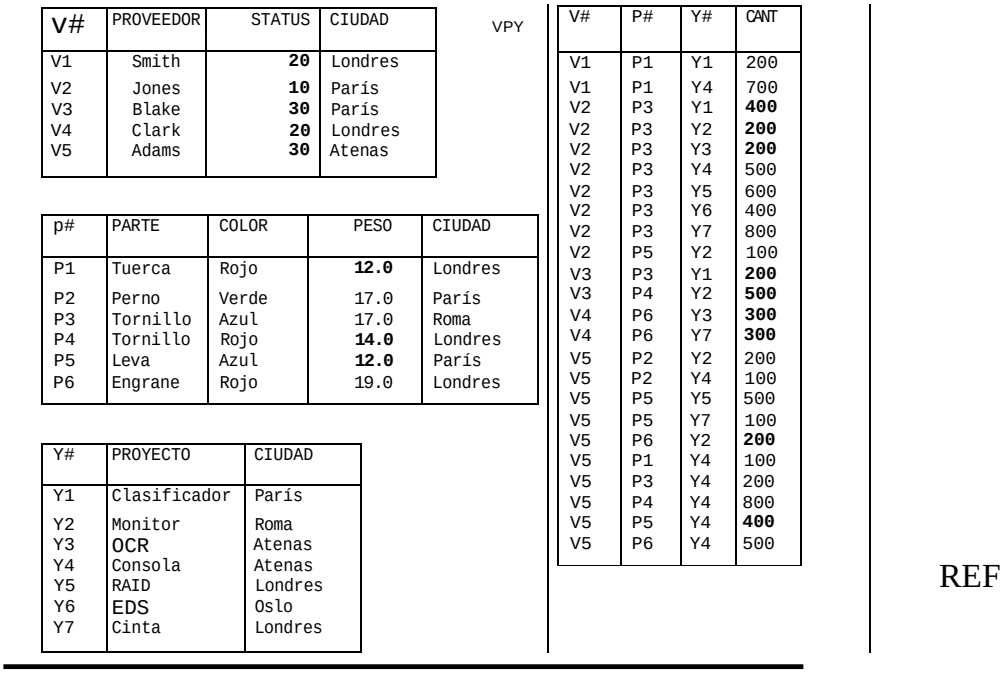
4.1 La figura 4.5 muestra algunos ejemplos de valores de datos para una forma ampliada de la base de datos de proveedores y partes, llamada base de datos de proveedores, partes y proyectos. Los proveedores (V), partes (P) y proyectos (Y) están identificados de manera única por el número de proveedor (V#), el número de parte (P#) y el número de proyecto (Y#), respectivamente. El signifi cado de una fila VPY (envío) es que el proveedor especificado suministra la parte especificada al proyecto especificado en la cantidad especificada (y la combinación de V#-P#-Y# identifica de ma nera única a esa fila). Escriba un conjunto apropiado de definiciones de SQL para esta base de datos. *Nota:* Esta base de datos se usará como apoyo de numerosos ejercicios en los capítulos subsiguientes.

4.2 En la sección 4.2 describimos la instrucción CREATE TABLE como la define el estándar de SQL por sí mismo. Sin embargo, muchos productos comerciales de SQL soportan opciones adi cionales para esa instrucción, que por lo regular tienen que ver con índices, asignación de espacio en disco y otros asuntos de implementación (minando con ello los objetivos de independencia física de los datos y de compatibilidad entre sistemas). Analice cualquier producto SQL que tenga disponible. ¿Se aplica a ese producto la crítica previa? Específicamente ¿qué opciones adicionales maneja ese producto para CREATE TABLE?

4.3 Analice una vez más cualquier producto SQL que tenga disponible. ¿Maneja ese producto el Es quema de información? Si no, ¿cómo luce su soporte del catálogo?

4.4 Dé formulaciones SQL para las siguientes actualizaciones a la base de datos de proveedores, partes y proyectos:

a. Insertar un nuevo proveedor V10 en la tabla V. El nombre y la ciudad son Smith y Nueva York, respectivamente. El status aún no se conoce.



V#

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	**20**	Londres
V2	Jones	**10**	París
V3	Blake	**30**	París
V4	Clark	**20**	Londres
V5	Adams	**30**	Atenas

VPY

V#	P#	Y#	CANT
V1	P1	Y1	200
V1	P1	Y4	700
V2	P3	Y1	**400**
V2	P3	Y2	**200**
V2	P3	Y3	**200**
V2	P3	Y4	500
V2	P3	Y5	600
V2	P3	Y6	400
V2	P3	Y7	800
V2	P5	Y2	100
V3	P3	Y1	**200**
V3	P4	Y2	**500**
V4	P6	Y3	**300**
V4	P6	Y7	**300**
V5	P2	Y2	200
V5	P2	Y4	100
V5	P5	Y5	500
V5	P5	Y7	100
V5	P6	Y2	**200**
V5	P1	Y4	100
V5	P3	Y4	200
V5	P4	Y4	800
V5	P5	Y4	**400**
V5	P6	Y4	500

p#	PARTE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	**12.0**	Londres
P2	Perno	Verde	17.0	París
P3	Tornillo	Azul	17.0	Roma
P4	Tornillo	Rojo	**14.0**	Londres
P5	Leva	Azul	**12.0**	París
P6	Engrane	Rojo	19.0	Londres

Y#	PROYECTO	CIUDAD
Y1	Clasificador	París
Y2	Monitor	Roma
Y3	OCR	Atenas
Y4	Consola	Atenas
Y5	RAID	Londres
Y6	EDS	Oslo
Y7	Cinta	Londres

REF

Figura 4.5 La base de datos de proveedores, partes y proyectos (valores de ejemplo

b. Cambiar el color de todas las partes rojas por naranja.

c. Eliminar todos los proyectos para los cuales no haya envíos.

4.5 Usando de nuevo la base de datos de proveedores, partes y proyectos, escriba un prograr instrucciones de SQL incrustado para listar todas las filas de proveedores, ordenándolas por n de proveedor. En el listado, cada proveedor debe estar seguido inmediatamente por todas las filas de
proyectos que abastece ese proveedor, ordenándolas por número de proyecto.

4.6 Sea la siguiente la definición de las tablas PARTES y ESTRUCTURA_PARTES:

```
CREATE TABLE PARTES
     ( P# ... , DESCRIPCION ... ,
       PRIMARY KEY ( P# ) ) ;

CREATE TABLE ESTRUCTURA_PARTES
     ( P#_MAY0R . . . , P#_MEN0R . . . , CANT ... , PRIMARY
       KEY ( P#_MAYOR, P#_MENOR ) , FOREIGN KEY (
       P#_MAYOR ) REFERENCES PARTES, FOREIGN KEY (
       P#_MENOR ) REFERENCES PARTES ) ;
```

La tabla ESTRUCTURA_PARTES muestra qué partes (P#_MAY0R) contienen otras [(P#_MENOR) como componentes del primer nivel. Escriba un programa SQL para listar todas las

ESTRUCTURA_PARTES

P#_MAYOR	P#_MEN0R	CANT
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

Figura 4.6 La tabla ESTRUCTURA_PARTES (valores de ejemplo).

partes componentes de una parte dada, en todos los niveles (el problema de **explosión de partes).** *Nota:* Los datos de ejemplo que muestra la figura 4.6 podrían ayudarle a visualizar este problema. Subrayamos que la tabla ESTRUCTURA_PARTES muestra la forma típica en que los datos de la *lista de materiales* —vea el capítulo 1, sección 1.3, subsección "Entidades y vínculos"— son representados en un sistema relational.

REFERENCIAS Y BIBLIOGRAFÍA

4.1 M. M. Astrahan y R. A. Lorie: "SEQUEL-XRM: A Relational System", Proc. ACM Pacific Re gional Conf., San Francisco, Calif, (abril, 1975).

Describe la primera implementación prototipo de SEQUEL, la primera versión de SQL [4.8]. Vea también las referencias [4.2—4.3], que realizan una función análoga para el System R.

4.2 M. M. Astrahan *et al.*: "System R: Relational Approach to Database Management", *A CM TODS 1,* No. 2 (junio, 1976).

System R fue la principal implementación prototipo del lenguaje SEQUEL/2 (más adelante SQL) [4.8]. Este artículo describe la arquitectura de System R como se planeó originalmente. Vea también la referencia [4.3].

4.3 M. W.Blasgen et al.:"System R: An Architectural Overview",IBM Sys. J.20, No. 1 (febrero, 1981).

Describe la arquitectura del System R como quedó al momento de implementar por completo el sistema (compárese con la referencia [4.2]).

4.4 Stephen Cannan y Gerard Otten: *SQL--The Standard Handbook.* Maidenhead, UK: McGraw-Hill International (1993).

"[Nuestro] objetivo... es proporcionar un trabajo de consulta que explique y describa [al SQL/92 como se definió originalmente] en una forma mucho menos formal y mucho más legible que el estándar mismo" (tomado de la introducción del libro).

4.5 Joe Celko: *SQL for Smarties: Advanced SQL Programming.* San Francisco, Calif.: Morgan Kaufmann (1995).

"Éste es el primer libro de SQL avanzado que ofrece una amplia presentación de las técnicas necesarias para apoyar sus avances desde un usuario ocasional de SQL a un programador experto" (tomado de la cubierta del libro).

4.6 Andrew Eisenberg y Jim Melton: "SQLJ Part 0, Now Known as SQL/OLB (Object Language Bindings)", *ACM SIGMOD Record 27,* No. 4 (diciembre, 1998). Vea también Gray Clossman *et al.* "Java and Relational Databases: SQLJ", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

4.7 Don Chamberlin: *Using the New DB2.* San Francisco, Calif.: Morgan Kaufmann (1996).

Una descripción legible y amplia de un producto SQL comercial de primera línea, por uno de los dos principales diseñadores del lenguaje SQL original [4.8].

Nota: El libro también expone "algunas decisiones controversiales" que se hicieron en el diseño de SQL; principalmente (a) la decisión de soportar valores nulos y (b) la decisión de permitir filas duplicadas. "Mi motivo [dice Chamberlin]... más que persuasivo, es histórico; reconozco que los nulos y los duplicados son aspectos religiosos... En su mayoría, los diseñadores [de SQL] más que teóricos eran personas prácticas y esta orientación se reflejó en muchas decisiones [de diseño]" Esta postura es muy diferente a la mía. Los nulos y duplicados son aspectos *científicos,* no religiosos; en este libro son expuestos de manera científica (en los capítulos 18 y 5, respectivamente Por lo que se refiere a "más que teóricos... personas prácticas", rechazamos categóricamente la sugerencia de que la teoría no es práctica; ya establecimos (en la sección 4.5) nuestra posición de que por lo menos, la teoría relacional es en realidad muy práctica.

4.8 Donald D. Chamberlin y Raymond F. Boyce: "SEQUEL: A Structured English Query Langua ge", Proc. 1974 ACM SIGMOD Workshop on Data Description, Access, and Control, Ann Arbor. Mich, (mayo, 1974).

El artículo que presentó por primera vez el lenguaje SQL (o SEQUEL, como se llamaba originalmente; el nombre se cambió después por razones legales).

4.9 Donald D. Chamberlin *et al:* "SEQUEL/2: A Unified Approach to Data Definition. Manipula tion, and Control", *IBM J. R&D. 20,* No. 6 (noviembre, 1976). Vea también las erratas en *IBM J. R&D. 21,* No. 1 (enero, 1977).

La experiencia de la implementación del primer prototipo de SEQUEL (expuesta en la referencia [4.1]) y los resultados de las pruebas de utilización reportados en la referencia [4.28], condujeron al diseño de una versión revisada del lenguaje, que se llamó SEQUEL/2. El lenguaje soportado por el System R [4.2-4.3] fue básicamente SEQUEL/2 (con la conspicua ausencia de las tan mencionadas propiedades de "aserción" y "disparador"; vea el capítulo 8), más ciertas ampliaciones sugeridas por la experiencia de los primeros usuarios [4.10].

4.10 Donald D. Chamberlin: "A Summary of User Experience with the SQL Data Sublanguage". Proc. Int. Conf. on Databases, Aberdeen, Scotland (julio, 1980). También disponible como IBM Research Report RJ2767 (abril, 1980).

Expone las experiencias de los primeros usuarios con el System R y propone algunas ampliacionnes al lenguaje SQL a la luz de estas experiencias. Algunas de estas ampliaciones —EXISTS. LIKE, PREPARE y EXECUTE— de hecho fueron implementadas en la versión final de System R. ***Nota*** Vea el capítulo 7 y el apéndice A para una explicación de EXISTS y LIKE, respectivamente.

4.11 Donald D. Chamberlin *et al.*: "Support for Repetitive Transactions and *Ad Hoc* Queries in Sys tem R", *ACM TODS 6,* No. 1 (marzo, 1981).

Ofrece algunas medidas de rendimiento del System R en los ambientes tanto de consulta como de "transacción enlatada". (Una "transacción enlatada" es una aplicación sencilla que accede sólo a una parte reducida de la base de datos y se compila antes del tiempo de ejecución. Corresponde a lo que nosotros llamamos *petición planeada* en el capítulo 2, sección 2.Í

mediciones se hicieron sobre un Sistema 370 de IBM, modelo 158, ejecutando System R bajo el sistema operativo VM. Se les describe como "preliminares"; sin embargo, con esta advertencia el artículo muestra, entre otras cosas, que (a) la compilación casi siempre es superior a la interpretación (incluso para consultas *ad hoc)* y (b) un sistema como System R es capaz de procesar varias transacciones enlatadas por segundo (en el supuesto que existan en la base de datos los índices adecuados).

Vale la pena destacar el artículo ya que fue uno de los primeros en desmentir la afirmación, muy común en ese momento, de que "los sistemas relacionales nunca tendrán un buen rendimiento". Por supuesto, desde que fueron publicados, los productos relacionales comerciales han alcanzado tasas de transacción en los cientos e incluso en los miles de transacciones por segundo.

4.12 Donald D. Chamberlin *et al.:* "A History and Evaluation of System R", CACM 24, No. 10 (octubre, 1981).

Describe las tres fases principales del proyecto System R (prototipo preliminar, prototipo multiusuario y evaluación), haciendo énfasis en las tecnologías de compilación y optimización que fueron pioneras en System R. Hay cierto entrecruzamiento entre este artículo y la referencia [4.13].

4.13 Donald D. Chamberlin, Arthur M. Gilbert y Robert A. Yost: "A History of System R and SQL / Data System", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, Francia (septiembre, 1981).

Expone las lecciones aprendidas del prototipo del System R y describe la evolución de ese prototipo en el primero de los productos de la familia DB2 de IBM, SQL/DS (nombre que cambió posteriormente a "DB2 para VM y VSE").

4.14 C. J. Date: "A Critique of the SQL Database Language", *ACM SIGMOD Record 14,* No. 3 (noviembre, 1984). Reeditado en *Relational Database: Selected Writings.* Reading, Mass.: Addison-Wesley (1986).

Como señalé en el capítulo, SQL no es perfecto. Este documento presenta una análisis crítico de varias de las principales deficiencias del lenguaje (en particular, desde el punto de vista de los lenguajes de cómputo formales en general, más que de los lenguajes de base de datos en particular). *Nota:* Parte de las críticas de este artículo no se aplican al SQL/92.

4.15 C. J. Date: "What's Wrong with SQL?", *en Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Expone algunas deficiencias adicionales de SQL, además de las identificadas en la referencia [4.14], bajo los títulos "What's Wrong with SQL *per se",* "What's Wrong with the SQL Standard" y "Application Portability". *Nota:* Una vez más, parte de las críticas de este documento no se aplican al SQL/92.

4.16 C. J. Date: "SQL Dos and Don'ts", en *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Este artículo ofrece algunos consejos prácticos sobre cómo utilizar SQL a fin de (a) evitar algunos de los errores potenciales que se derivan de los problemas expuestos en las referencias [4.14-4.15] y [4.18] y (b) apreciar los máximos beneficios posibles en términos de productividad, portabilidad, etcétera.

4.17 C. J. Date: "How We Missed the Relational Boat", en *Relational Database Writings 1991-1994.* Reading, Mass.: Addison-Wesley (1995).

Un resumen conciso de las deficiencias de SQL con respecto al apoyo (o a la falta de apoyo) que da a los aspectos estructurales, de integridad y de manipulación del modelo relacional.

4.18 C. J. Date: "Grievous Bodily Harm" (en dos partes), *DBP&D 11*, No. 5 (mayo, 1998) y //, No. 6 (junio, 1998); "Fifty Ways to Quote Your Query", en el sitio web *DBP&D www.dbpd.com* (julio, 1998).

SQL es un lenguaje en extremo redundante, en el sentido de que hasta las consultas más triviales pueden expresarse en muchas formas diferentes. Los artículos ilustran esta idea y explican algunas de sus implicaciones. En particular, muestran que la cláusula GROUP BY, la cláusula HAVING y las variables de rango podrían eliminarse del lenguaje sin pérdida de funcionalidad (lo mismo se aplica a la construcción "IN *subconsulta"). Nota:* Todas estas construcciones de SQL se explican en el capítulo 7 (sección 7.7) y en el apéndice A.

4.19 C. J. Date y Hugh Darwen: *A Guide to the SQL Standard* (4a edición). Reading, Mass.: Addison-Wesley (1997).

Un amplio tutorial sobre SQL/92, que incluye CLI y PSM. En particular, el libro contiene un apéndice (el apéndice D) que documenta "muchos aspectos del estándar que parecen no estar definidos en forma adecuada o incluso estar definidos incorrectamente". *Nota:* Las referencias [4.4] y [4.27] también son tutoriales de SQL/92.

4.20 C. J. Date y Colin J. White: *A Guide to DB2* (4a. edición). Reading, Mass.: Addison-Wesley (1993).

Ofrece un repaso amplio y minuciosos del producto DB2 original de IBM (en 1993) y algunos de los productos que le acompañan. DB2, al igual que SQL/DS [4.13], se basó en System R.

4.21 Neal Fishman: "SQL *du Jour", DBP&D 10*, No. 10 (octubre, 1997).

Un examen deprimente de algunas de las incompatibilidades a encontrar entre los productos SQL que afirman todos ellos que "manejan el estándar de SQL".

4.22 International Organization for Standardization (ISO): *Database Language SQL*, Document ISO/IEC 9075:1992. También disponible como American National Standards Institute (ANSI) Docu ment ANSI X3.135-1992.

La definición original ISO/ANSI SQL/92 (conocida por los entendidos como *ISO/IEC 9075* o en ocasiones solamente como *¡SO 9075).* El documento original, de un solo tomo, se ha ampliado desde entonces en una serie abierta de partes independientes, bajo el título general de *Information Technology —Database Languages— SQL.* Al momento de la publicación de este libro, se habían definido las siguientes partes (aunque ciertamente no todas terminadas):

Parte 1: Framework (SQL/Framework)
Parte 2: Foundation (SQL/Foundation)
Parte 3: Call-Level Interface (SQL/CLI)
Parte 4: Persistent Stored Modules (SQL/PSM)
Parte 5: Host Language Bindings (SQL/Bindings)
Parte 6: XA Specialization (SQL/Transaction)
Parte 7: Temporal (SQL/Temporal)
Parte 8: *There is no Part 8*
Parte 9: Management of External Data (SQL/MED)
Parte 10: Object Language Bindings (SQL/OLB)

Las propuestas del SQL3 que esperamos se ratifiquen en 1999, pertenecen lógicamente a las partes 1, 2, 4 y 5. Los borradores que definen dichas propuestas se pueden encontrar en la World Wide Web en *ftp://jerry.ece.umassd.edu/isowg3/dbl/BASEdocs/public.*

Nota: La idea que vale la pena mencionar es que, aunque SQL es ampliamente reconocido como el estándar internacional de base de datos "relacional", el documento del estándar no se

describe a sí mismo como tal; de hecho, ¡nunca emplea realmente el término "relación"! (Como mencioné en un pie de página anterior, tampoco menciona el término "base de datos").

4.23 International Organization for Standardization (ISO): *Information Technology—Database Languages—SQL-Technical Corrigendum 2*, Document ISO/IEC 9075:1992/Cor.2:1996(E).

Contiene una gran cantidad de revisiones y correcciones a la versión original de la referencia [4.22]. Por desgracia, dichas revisiones y correcciones no arreglan casi ninguno de los problemas identificados en la referencia [4.19].

4.24 Raymond A. Lorie y Jean-Jacques Daudenarde: *SQL and Its Applications.* Englewood Cliffs, N.J.: Prentice-Hall (1991).

Un libro "práctico" sobre SQL (casi la mitad del libro consiste en una serie de casos de estudio detallados que comprenden aplicaciones realistas).

4.25 Raymond A. Lorie y J. F. Nilsson: "An Access Specification Language for a Relational Data Base System", *IBM J. R&D. 23*, No. 3 (mayo, 1979).

Ofrece más detalles sobre un aspecto específico del mecanismo de compilación de System R [4.11,4.25-4.26]. Para toda instrucción SQL dada, el optimizador de System R genera un programa en un lenguaje interno denominado ASL (Lenguaje de Especificación de Acceso). Este lenguaje sirve como interfaz entre el optimizador y el *generador de código*. (El generador de código, como su nombre lo indica, convierte un programa ASL en código de máquina.) ASL consiste de operadores como "scan" e "insert" sobre objetos como índices y archivos almacenados. La finalidad de ASL era hacer más manejable el proceso general de traducción, dividiéndolo en un conjunto de subprocesos bien definidos.

4.26 Raymond A. Lorie y Bradford W. Wade: "The Compilation of a High-Level Data Language", IBM Research Report RJ2598 (agosto, 1979).

System R presentó un esquema para compilar consultas antes del tiempo de ejecución y recompilarlas posteriormente en forma automática si en el transcurso había cambiado de manera importante la estructura de la base de datos. Este artículo describe con cierto detalle el mecanismo de compilación y recompilación de System R; sin embargo, no trata las cuestiones de la optimización (para información sobre este último tema, vea la referencia [17.34]).

4.27 Jim Melton y Alan R. Simon: *Understanding the New SQL: A Complete Guide.* San Mateo, Calif.: Morgan Kaufmann (1993).

Un tutorial sobre SQL/92 (tal como se definió originalmente). Melton fue el editor de la especificación original del SQL/92 [4.22].

4.28 Phyllis Reisner, Raymond F. Boyce y Donald D. Chamberlin: "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL", Proc. NCC 44, Anaheim, Calif. Montvale, N.J.: AFIPS Press (mayo, 1975).

SEQUEL [4.8], el antecesor de SQL, estaba basado en un lenguaje anterior denominado SQUARE. De hecho, ambos lenguajes eran básicamente iguales, aunque SQUARE usaba una sintaxis más bien matemática, mientras que SEQUEL estaba basado en palabras del inglés como SELECT, FROM, WHERE, etcétera. El artículo informa sobre una serie de experimentos que se realizaron sobre la utilización de los dos lenguajes con estudiantes universitarios como sujetos. Como resultado de este trabajo se hicieron varias revisiones a SEQUEL [4.9].

4.29 David Rozenshtein, Anatoly Abramovich y Eugene Birger: *Optimizing Transact-SQL: Advanced Programming Techniques.* Fremont, Calif.: SQL Forum Press (1995).

Transact-SQL es el dialecto de SQL soportado por los productos Sybase y SQL Server. Este libro presenta una serie de técnicas de programación para Transact-SQL con base en el uso *de. fun-*

clones características (definidas por los autores como "dispositivos que permiten a los programadores codificar la lógica condicional como... expresiones dentro de las cláusulas SELECT, WHERE, GROUP BY y SET"). Aunque expresadas específicamente en términos de Transact-SQL, las ideas tienen en realidad una aplicación más amplia. *Nota:* Tal vez debamos agregar que la "optimización" que se menciona en el título del libro no se refiere al componente optimizador del DBMS, sino más bien a "optimizaciones" que los propios usuarios pueden hacer en forma manual.

RESPUESTAS A EJERCICIOS SELECCIONADOS

4.1

```
CREATE TABLE V
    ( V#          CHAR (5),
      PROVEEDOR   CHAR (20),
      STATUS      NUMERIC(5),
      CIUDAD      CHAR(15),
     PRIMARY KEY  ( V# ) ) ;

CREATE TABLE P
    ( P#          CHAR (6),
      PARTE       CHAR (20),
      COLOR       CHAR(6),
      PESO        NUMERIC(5,1)
      CIUDAD      CHAR(15),
     PRIMARY KEY   ( P# ) ) ;

CREATE TABLE Y
     ( Y#          CHAR (4) ,
       PROYECTO    CHAR (20),
       CIUDAD      CHAR(15),
      PRIMARY KEY   ( Y# ) ) ;

CREATE TABLE VPY
     ( V#         CHAR (5)  ,
       P#         CHAR   (6),
       Y#         CHAR   (4),
       CANT       NUMERIC(9),
      PRIMARY KEY : V#, P#, Y# ),
      FOREIGN KEY   V# ) REFERENCES V,
      FOREIGN KEY   P# ) REFERENCES P,
      FOREIGN KEY   Y# ) REFERENCES Y )
```

```
4.4 a. INSERT INTO V ( V#, PROVEEDOR, CIUDAD )
       VALUES ( 'V10',  'Smith',  'Nueva York' ) ;
```

Aquí, a STATUS se le asigna el valor predeterminado aplicable.

```
b. UPDATE P
   SET   COLOR = 'Naranja'
   WHERE COLOR = 'Rojo' ;

C. DELETE
   FROM  Y
   WHERE Y# NOT IN (
        SELECT Y# FROM
        VPY ) ;
```

Observe la subconsulta anidada y el operador IN (en realidad, el operador IN *negado)* en la solución a la parte c. Vea el capítulo 7 para una explicación más amplia.

4.5 Observe que podría haber algunos proveedores que no suministren proyecto alguno; la siguiente solución maneja en forma satisfactoria a estos proveedores (¿exactamente cómo?). Primero, definimos dos cursores, CV y CY, como sigue:

```
EXEC SQL DECLARE CV CURSOR FOR
         SELECT V.V#, V.PROVEEDOR, V.STATUS, V.CIUDAD
         FROM  V ORDER BY V# ;

EXEC SQL DECLARE CY CURSOR FOR
         SELECT Y.Y#, Y.PROYECTO, Y.CIUDAD
         FROM  Y WHERE Y.Y# IN
              ( SELECT VPY.Y#
                FROM  VPY
                WHERE VPY.V# • :CV_V# )
         ORDER BY Y# ;
```

(Observe, una vez más, la subconsulta anidada y el operador IN.)

Cuando se abre el cursor CY, la variable anfitrión CV_V# contendrá un valor de número de proveedor, al que se tiene acceso a través del cursor CV. La lógica en forma de procedimiento es básicamente la siguiente:

```
EXEC SQL OPEN CV ;
DO para todas las filas de V accesibles mediante CV ;
   EXEC SQL FETCH CV INTO :CV_V#, :CV_PR, :CV_ST, :CV_CD ;
   imprime CV_V#, Cv_PR, CV_ST, CV_CD ;
   EXEC SQL OPEN CY ;
   DO para todas las filas de Y accesibles mediante CY ;
      EXEC SQL FETCH CY INTO :CY_Y#, :CY_PY, :CY_CD ;
      imprime CY_Y#, CY_PY, CY_CD ;
   END DO ;
   EXEC SQL CLOSE CY ;
END DO ; EXEC SQL CLOSE
CV ;
```

4.6 Éste es un buen ejemplo de un problema que SQL/92 no maneja bien. La dificultad básica es Ja siguiente: Necesitamos "explotar" la parte dada a *n* niveles, donde el valor de *n* se desconoce al momento de escribir el programa. Una forma comparativamente directa de realizar —si fuera posible— dicha "explosión" a nivel *n*, sería por medio de un programa recursivo, en el cual, cada invocación recursiva crea un nuevo cursor, como sigue:

```
CALL RECURSION ( P#DAD0 ) ;

RECURSION: PROC ( P#_SUPERIOR ) RECURSIVE ;
    DCL P#_SUPERIOR . . .   ; DCL P#_INFERIOR
    . . .  ; EXEC SQL DECLARE C CURSOR
    "reabrible" FOR
             SELECT P#_MEN0R
             FROM  ESTRUCTURA_PARTES
             WHERE P#_MAY0R = :P#_SUPERIOR ;

    imprime P#_SUPERIOR ;
    EXEC SQL OPEN C ;
    DO para todas las filas de ESTRUCTURA_PARTES accesibles mediante C ;
       EXEC SQL FETCH C INTO :P#_INFERIOR ;
       CALL RECURSION ( P#_INFERIOR ) ;
    END DO ;
    EXEC SQL CLOSE C ;
END PROC ;
```

Aquí damos por hecho que la especificación (ficticia) "factible de ser reabierta" en DECLARE CURSOR significa que es válido abrir ese cursor aun cuando ya estaba abierto y que el efecto de dicho OPEN es crear un nuevo ejemplar del cursor para la expresión de tabla especificada (usando los valores actuales de toda variable anfitrión a la que se hace referencia en esa expresión). Damos también por hecho que las referencias a dicho cursor en FETCH (entre otros) son referencias al ejemplar "actual" y que CLOSE destruye ese ejemplar y restablece el ejemplar anterior como el "actual". En otras palabras, damos por hecho que un cursor factible de ser reabierto forma una *pila* y que OPEN y CLOSE sirven como los operadores "push" y "pop" de esa pila.

Por desgracia, en la actualidad estas suposiciones son puramente hipotéticas. En el SQL actual no existe algo similar a un cursor factible de ser reabierto (de hecho, cualquier intento de abrir un cursor ya abierto, fracasará). El código precedente no es válido. Pero el ejemplo aclara que los "cursores factibles de ser reabiertos" serían una ampliación muy deseable.

Puesto que el enfoque anterior no funciona, ofrecemos un bosquejo de un enfoque posible (aunque muy ineficiente) que sí funciona.

```
CALL RECURSION  ( P#DAD0 ) ;

RECURSION: PROC ( P#_SUPERIOR ) RECURSIVE ;
    DCL P#_SUPERIOR ... ;
    DCL P#_INFERIOR . . . INITIAL ( '     ' ) ¡
    EXEC SQL DECLARE C CURSOR FOR
                     SELECT P# MENOR
                     FROM   ESTRUCTURA_PARTES
                     WHERE P#_MAY0R - :P#_SUPERIOR
                     AND   P#_MENOR > :P#_INFERIOR
                     ORDER BY P#_MENOR ;

    imprime P#_SUPERIOR ;
    DO "por siempre" ;
       EXEC SQL OPEN C ;
       EXEC SQL FETCH C INTO :P#_INFERIOR ¡
       EXEC SQL CLOSE C ¡
       IF ningún "P# inferior" recuperado THEN RETURN ; END IF ;
       IF "P# inferior" recuperado THEN
      CALL RECURSION ( P#_INFERIOR ) ; END IF ; END
DO ; END PROC ;
```

Observe que en esta solución se usa el mismo cursor en cada llamada a RECURSION. (En contraste, cada vez que se llama a RECURSION se crean de manera dinámica nuevas instancias *i* P#_SUPERIOR y P#_INFERIOR; dichas instancias se destruyen al terminar esa llamada.) Por esto, tenemos que usar un truco

```
. . .  AND P#_MENOR > :P#_INFERIOR ORDER BY P#_MEN0R
```

para que, en cada llamada a RECURSION, ignoremos todos los componentes inmediatos (los P#_INFERIOR) del P#_SUPERIOR actual que ya han sido procesados.

Consulte (a) las referencias [4.5] y [4.7] para la explicación de algún enfoque alternativo a este problema al estilo de SQL, (b) el capítulo 6 (al final de la sección 6.7) para la descripción de un operador relacional pertinente llamado *cierre transitivo,* y (c) el apéndice B para un repaso de algunas propiedades relevantes de SQL3.

PARTE II

EL MODELO RELACIONAL

El modelo relacional es sin lugar a dudas el fundamento de la tecnología moderna de base de datos; este fundamento es el que hace de este campo una ciencia. Entonces, todo libro sobre fundamentos de la tecnología de base de datos que no incluya una minuciosa cobertura del modelo relacional es, por definición, superficial. De igual forma, difícilmente se puede justificar cualquier afirmación de dominar el campo de las bases de datos si quien lo afirma no comprende a fondo el modelo relacional. Es necesario agregar que no es que el material sea en absoluto "difícil" —en realidad no lo es— sino que, para decirlo de nuevo, *son* los fundamentos y por lo que podemos ver, seguirán siéndolo.

Como expliqué en el capítulo 3, el modelo relacional se ocupa de tres aspectos principales de la información: la *estructura* de datos, la *manipulación* de datos y la *integridad* de los datos. En esta parte del libro consideraremos cada uno de estos aspectos: el capítulo 5 explica la estructura, los capítulos 6 y 7 explican la manipulación, y el capítulo 8 explica la integridad. (Hay dos capítulos sobre manipulación ya que esta parte del modelo puede ser concebida de dos formas distintas [aunque equivalentes] conocidas como *álgebra relacional* y *cálculo relacional*, respectivamente.) Por último, el capítulo 9 explica el importante tema de las *vistas*.

Ahora bien, es importante entender que el modelo relacional no es algo estático; a través de los años ha evolucionado y se ha extendido; y continúa haciéndolo.* El texto que sigue refleja mi forma de pensar y también la de otros estudiosos del tema (en particular, como mencioné en el prefacio, esta forma de pensar está influenciada completamente por las ideas de *The Third Manifesto* [3.3]). El tratamiento pretende ser bastante completo e incluso definitivo (al momento de la publicación de este libro), aunque desde luego es de estilo pedagógico; sin embargo, usted no deberá tomar lo que sigue como la última palabra sobre el tema.

Para repetir, el modelo relacional no es difícil de entender, pero es una teoría y la mayoría de ellas vienen equipadas con su propia terminología y (por las razones ya señaladas en la sección 3.3) el modelo relacional no es la excepción a este respecto. Y por supuesto, en esta parte del libro usaremos esa terminología especial. Sin embargo, no puede negarse que la terminología puede resultar en principio desconcertante y de hecho puede representar un obstáculo para la comprensión (este último hecho es en particular desafortunado, ya que las ideas subyacentes no son difíciles en absoluto). De modo que si tiene problemas para comprender parte del material siguiente, por favor tenga paciencia. Probablemente descubrirá que los conceptos en realidad resultan ser bastante directos, una vez que se haya familiarizado con la terminología.

---

* En este aspecto se asemeja a las matemáticas (que tampoco son estáticas sino que se desarrollan con el paso del tiempo). De hecho, el propio modelo relacional podría ser concebido como una pequeña rama de las matemáticas.

Como pronto verá, los cinco capítulos siguientes son muy extensos (esta parte es casi un libro por derecho propio). Pero la extensión refleja la importancia del tema en cuestión. Sena posible ofrecer un panorama general del tema en tan sólo una o dos páginas; de hecho, una de las principales ventajas del modelo relacional es que sus ideas básicas pueden explicarse y comprenderse con mucha facilidad. Sin embargo, un tratamiento de una o dos páginas no puede hacer justicia al tema, ni ilustrar su amplio rango de aplicabilidad. Entonces, la gran extensión de esta parte del libro debe ser considerada no como un comentario sobre la complejidad del modelo, sino como un tributo a su importancia y a su éxito como base de numerosos desarrollos de largo alcance.

Por último, un comentario con respecto a SQL. En la parte I ya he explicado que SQL es el lenguaje estándar de las bases de datos "relacionales" y que prácticamente todos los productos de bases de datos del mercado lo manejan (o, para ser más precisos, algún dialecto de él [4.21]). Como consecuencia, ningún libro moderno de base de datos estaría completo sin una amplia cobertura de SQL. Por lo tanto, los capítulos que cubren otros aspectos del modelo relacional también explican, donde corresponde, las características relevantes de SQL (están basados en el capítulo 4, que cubre los conceptos básicos de SQL).

CAPITULO 5

Dominios, relaciones y varrels base

5.1 INTRODUCCIÓN

Como expliqué en el capítulo 3, podemos considerar que el modelo relacional tiene tres partes principales, que tienen que ver con la *estructura de datos*, la *integridad de los datos* y la *manipulación de datos*, respectivamente. Cada parte tiene su propia terminología especial. Los términos **estructurales** más importantes se ilustran en la figura 5.1 (que, como puede ver, están basados en la relación de proveedores de la base de datos de proveedores y partes de la figura 3.8, la cual fue ampliada para mostrar los tipos de datos o dominios aplicables). Los términos en cuestión son: el propio término *relación* (por supuesto), *tupia, cardinalidad, atributo, grado, dominio* y *clave primaria.*

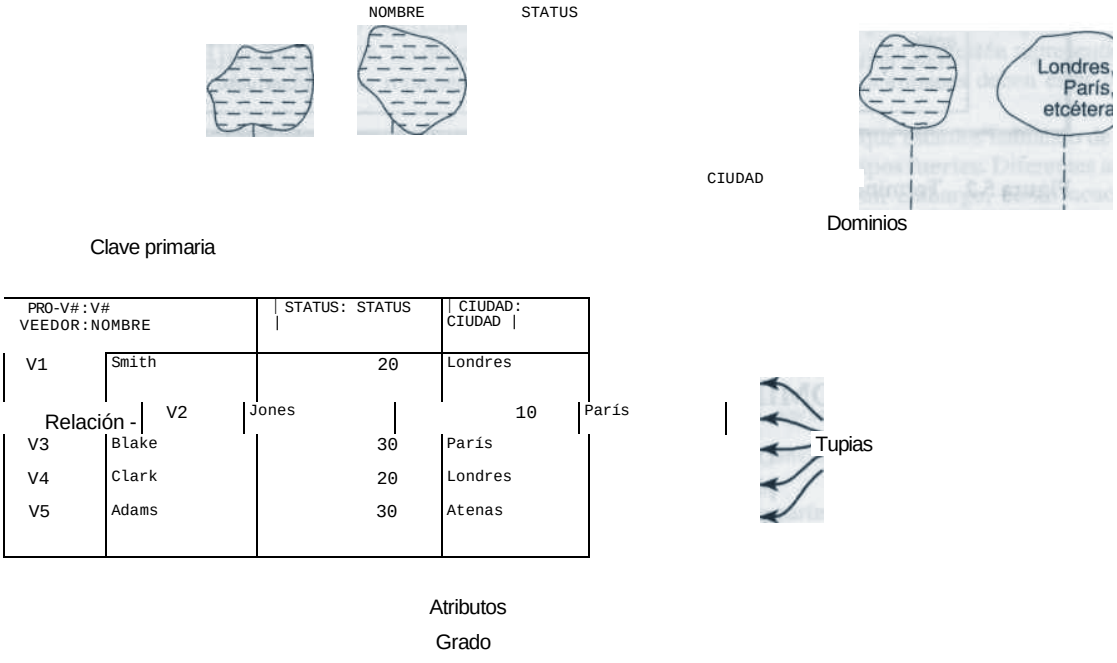


Figura 5.1 Terminología estructural.

De estos términos, al menos *relación* y *clave primaria* deben serle bastante familiares po: lo visto en el capítulo 3; en este capítulo explicaremos los demás términos de manera inform y después, en los capítulos subsiguientes, daremos definiciones más formales. Para ser breves. si pensamos en una relación como una tabla, entonces una **tupia** corresponde a una fila de dicha tabla y un **atributo** a una columna; al número de tupias se le llama **cardinalidad** y al número de atributos se le denomina **grado; y** un **dominio** es un conjunto de valores, de donde se toman los valores de atributos específicos de relaciones específicas. Por ejemplo, el dominio etiquetad como V# en la figura 5.1 es el conjunto de todos los números de proveedor posibles y cada valor V# que aparece en la relación de proveedores es algún valor de ese conjunto (en forma similar, cada valor V# que aparece en la relación de envíos —vea la figura 3.8— también es algún valoi de ese conjunto).

La figura 5.2 presenta un resumen de lo anterior. Sin embargo, debe entender que las "equivalencias" que muestra esa figura sólo son aproximadas (los términos relacionales formak tienen definiciones precisas, mientras que sus "equivalentes" informales sólo son definiciones burdas). De ahí que, por ejemplo, aunque una relación y una tabla en realidad no sean lo mismo —como vimos en la parte I de este libro— en la práctica es común pretender que lo son.

Pasemos ahora a nuestro tratamiento formal.

Término relacional formal	*Equivalente informal*
relación	tabla
tupia	fila o registro
cardinalidad	número de filas
atributo	columna o campo
grado	número de columnas
clave primaria	identificador único
dominio	conjunto de valores válidos

Figura 5.2 Terminología estructural (resumen).

5.2 DOMINIOS

Un dominio no es mas que un **tipo de datos** (para abreviar, un *tipo);* posiblemente un tipo simple **definido por el sistema** como INTEGER o CHAR o, en forma más general, un tipo definido **por el usuario** como V# o P# o PESO o CANT en la base de datos de proveedores y partes. De hecho, podemos usar los términos *tipo y dominio* de manera indistinta y así lo haremos en este libro (aunque preferimos el término *tipo;* cuando usemos el término *dominio,* lo haremos principalmente por razones históricas).

Pero, ¿qué es un tipo? Entre otras cosas, es **un conjunto de valores** —*todos los valores posibles* del tipo en cuestión. Por ejemplo, el tipo INTEGER es el conjunto de todos los entera posibles; el tipo V# es el conjunto de todos los números de proveedor posibles: y así sucesivamente. Además, junto con la noción de un tipo dado está asociada la noción de los operadores válidos que se pueden aplicar legalmente a valores de ese tipo; es decir, se puede operar *exclu-*

sivamente sobre valores de ese tipo por medio de los operadores definidos para el mismo. Por ejemplo, en el caso del tipo INTEGER (el cual, por razones de simplicidad, lo tomamos como definido por el sistema):

- El sistema proporciona los operadores "=", "<", etcétera, para comparar enteros;
- Además proporciona los operadores "+", "*", etcétera, para realizar operaciones aritméti cas sobre los enteros;
- *No* proporciona los operadores "II" (concatenar), SUBSTR (subcadena), etcétera, para rea lizar operaciones de cadena sobre los enteros. En otras palabras, las operaciones de cadena en enteros no son soportadas.

Y en un sistema que brinde soporte adecuado de tipos, podríamos definir nuestros propios tipos: por ejemplo, el tipo V#. Y probablemente podríamos definir operadores "=", "<", etcétera, para comparar los números de proveedores. Sin embargo, probablemente *no* definiríamos operadores "+", "*", etcétera, lo que significaría que para este tipo no se manejaría aritmética sobre los números de proveedor (¿por qué querríamos sumar o multiplicar dos números de proveedor?).

Por lo tanto, observe que distinguimos con mucho cuidado entre un **tipo** *por sí mismo* y por otra parte, la **representación física** de los valores de ese tipo dentro del sistema.* (De hecho, los tipos son un aspecto del *modelo,* mientras que las representaciones físicas son un aspecto de la *implementación.)* Por ejemplo, los números de proveedor pueden ser representados físicamente como cadenas de caracteres, pero eso no significa que podamos realizar operaciones de cadena de caracteres sobre los números de proveedor; más bien, sólo podemos realizar dichas operaciones si fueron definidos los operadores apropiados para el tipo. Y por supuesto, los operadores que definimos para un tipo determinado dependerán del *significado* o la *semántica* que se pretenden para el tipo en cuestión, no de la forma en que estén representados físicamente los valores de ese tipo; de hecho, esas representaciones físicas deben estar *ocultas* en lo que al usuario concierne.

A estas alturas, quizá se haya dado cuenta de que estamos hablando de lo que en los círculos de lenguajes de programación se conoce como **tipos fuertes.** Diferentes autores tienen definiciones ligeramente distintas para este término; sin embargo, como nosotros lo empleamos significa, entre otras cosas, que (a) todo valor *tiene* un tipo y que (b) siempre que tratemos de realizar una operación, el sistema comprueba que los operandos son del tipo correcto para la operación en cuestión. Por ejemplo, considere la siguiente expresión:

```
P.PESO + VP.CANT        /* el peso de la parte más la cantidad del envío */

P.PESO * VP.CANT        /* el peso de la parte por la cantidad del envío */
```

La primera expresión no tiene sentido y el sistema debe rechazarla. Por otra parte, la segunda sí tiene sentido; denota el peso total de todas las partes involucradas en el envío. De modo que los

---

* En la literatura, a los tipos en ocasiones se les llama tipos de datos **abstractos** (ADTs), para enfatizar la idea de que los tipos deben distinguirse de su implementación. Sin embargo, nosotros no usamos este término, ya que sugiere que podría existir algún tipo que no sea "abstracto" en este sentido, cuando creemos que *siempre* debe establecerse una distinción entre un tipo y su implementación.

operadores que definiríamos para pesos y cantidades (en combinación), incluirían presuntamente el "*" pero no el "+".

Aquí tenemos más ejemplos, que ahora involucran operaciones de comparación (en realidad, comparaciones de igualdad):

```
P.PESO • VP.CANT

P.CIUDAD = V.CIUDAD
```

De nuevo, la primera expresión no tiene sentido pero sí la segunda. Así que los operadores que definiríamos para pesos y cantidades (en combinación) presuntamente no incluirían el "= aunque los de ciudades sí lo incluirían.* (En realidad, nosotros seguimos la referencia [3.3] al insistir, con suficiente razón, que el operador de comparación de igualdad "=" se defina para *todo* tipo; ya que siempre debe ser posible comprobar si dos valores del mismo tipo son de hecho el mismo valor.)

Ahora, observe que hasta el momento no hemos dicho nada acerca de la naturaleza de los valores que constituyen un tipo. De hecho, pueden ser **de cualquier clase.** Tendemos a imaginarlos como muy simples —números, cadenas, etcétera— pero no hay absolutamente nada en el modelo relacional que requiera que se limiten a dichas formas simples. Por lo tanto, podemos tener dominios de grabaciones de sonido, de mapas, de grabaciones de vídeo, de dibujos de ingeniería, de planos arquitectónicos, de puntos geométricos (etcétera). El único requisito es, nuevamente, que los valores del dominio sean manipulables *exclusivamente* por medio de los operadores definidos para el dominio en cuestión (la representación física debe ocultarse).

El mensaje anterior es tan importante —y tan malinterpretado— que lo enunciaremos de nuevo en términos diferentes:

La cuestión referente a qué tipos de datos son soportados es ortogonal a la cuestión del soporte para el modelo relacional

---

Los valores tienen tipo

Todo valor tiene un tipo. En otras palabras, si v es un valor, entonces se puede pensar en v como si portara una bandera que anuncia "Soy un entero" o "Soy un número de proveedor"o "Soy un punto geométrico" (etcétera). Observe que, por definición, cualquier valor dado siempre será exactamente de un tipo y no podrá cambiar jamás su tipo. (De lo que se desprende que los tipos distintos son siempre *disjuntos,* lo que significa que no tienen valores en común.)

---

* Por cierto, con respecto a la cuestión de decidir qué operadores son válidos para qué tipos, observamos que históricamente gran parte de la literatura de bases de datos —incluidas las primeras ediciones de este libro— consideraba sólo operadores de comparación como "=" y ">" e ignoraba otros operadores como"+" y"*".

[t]Excepto tal vez si se soporta herencia de tipos. Ignoraremos esta posibilidad hasta el capítulo 19.

Un tipo determinado puede ser *escalar* o *no escalar*. Un tipo **no escalar** es aquel que está definido explícitamente para tener componentes visibles para el usuario. En particular, los tipos de *relación* son no escalares en este sentido; por ejemplo, la relación que muestra la figura 5.1 es de un cierto tipo de relación y ese tipo en realidad tiene componentes visibles para el usuario (a saber, los atributos V#, PROVEEDOR, STATUS y CIUDAD). En contraste, los tipos **escalares** son aquellos que no tienen componentes visibles para el usuario. Los puntos que se desprenden son:

1. Para el resto de esta sección, todos los tipos que manejaremos serán específicamente es calares y por lo tanto, usaremos el término *tipo* (sin calificar) para referirnos al tipo escalar.
2. Por supuesto, la representación *física* de un valor escalar dado —es decir, un valor dado de un tipo escalar dado— puede ser arbitrariamente compleja. En particular, puede tener com ponentes (pero dichos componentes no serán visibles para el usuario). Por ejemplo, en cir cunstancias apropiadas, un valor escalar determinado podría tener una representación física consistente en un arreglo de pilas con listas de cadenas de caracteres.
3. En ocasiones se dice que los tipos escalares están **encapsulados,** ya que no tienen compo nentes visibles para el usuario. También se dice en ocasiones que son **atómicos.** Sin em bargo, preferimos no utilizar ninguno de estos términos, ya que han ocasionado mucha confusión en el pasado (para nosotros, como para cualquier otro). Para ser específicos, oca sionaron confusión (y a menudo la siguen causando) sobre las distinciones entre modelo e implementación y entre tipo y representación.
4. Como veremos dentro de poco, los tipos escalares sí tienen lo que se denomina *repre sentaciones posibles;* y éstas a su vez sí tienen componentes visibles para el usuario. No per mita que este hecho le confunda: los componentes en cuestión *no* son componentes del tipo, son componentes de la representación posible. El tipo en sí, sigue siendo escalar en el sen tido previamente definido.

Definición de tipos

Para ilustrar las ideas que estamos explicando, a lo largo de este libro usaremos el lenguaje **Tutorial D** (o, más bien, una ligera variante del mismo), el cual fue mencionado por primera vez en el capítulo 3. El **Tutorial D** podría ser caracterizado, a grandes rasgos, como un lenguaje al estilo de Pascal. Presentaremos cada una de sus características sobre la marcha. Obviamente, lo primero que necesitamos es una forma para que los usuarios definan sus propios tipos:

```
TYPE <nombre de tipo> <representación posible>  ...  ;
```

A manera de ejemplo, presentamos las definiciones de tipos (retomadas de la figura 3.9) para la base de datos de proveedores y partes:

```
TYPE V#     POSSREP ( CHAR ) ;
TYPE NOMBRE POSSREP ( CHAR ) ;
TYPE P#     POSSREP ( CHAR ) ;
TYPE COLOR  POSSREP ( CHAR ) ;
TYPE PESO   POSSREP ( RATIONAL ) ;
TYPE CANT   POSSREP ( INTEGER ) ;
```

Explicación:

1. Primero recuerde (del capítulo 3) que el atributo STATUS de proveedor y los atributos CIUDAD de proveedor y de parte están definidos en términos de tipos integrados, en lugar de estar definidos por el usuario; de manera que no se muestran definiciones correspondientes a estos atributos.
2. Como ya hemos visto, las representaciones físicas están ocultas para el usuario. Por lo tanto, las definiciones de tipos no dicen nada acerca de esas representaciones físicas. En vez de ello, dichas representaciones deben ser especificadas como parte de la transformación con ceptual/interna (vea el capítulo 2, sección 2.6).

   Sin embargo, por razones que explicaré en la siguiente subsección, sí requerimos que cada tipo tenga asociada por lo menos una representación posible. En el ejemplo, los valores de tipo V#, NOMBRE, P# y COLOR tal vez podrían ser representados como caden de caracteres (de cualquier longitud necesaria); los valores de tipo CANT podrían ser re-presentados como enteros, y los valores de tipo PESO como números racionales. *Nota: i* lo largo de este libro (siguiendo la referencia [3.3]), preferimos el término más preciso RATIONAL en lugar del más familiar REAL.
3. Adoptamos las convenciones obvias de sintaxis en donde (a) las representaciones] sin nombre heredan el nombre del tipo relevante; (b) los componentes de las represe ciones posibles sin nombre heredan el nombre de la representación relevante posible. 1 ahí que, por ejemplo, la única representación posible definida para el tipo CANT se llame también CANT, así como el único componente de esa representación posible.
4. La definición de un nuevo tipo ocasiona que el sistema cree una entrada en el catálogo paral describirlo (si necesita recordar algo con respecto al catálogo, consulte el capítulo 3, sec-j ción 3.6). Observaciones similares se aplican también a las definiciones de operadores (vea las siguientes dos subsecciones).
5. Quizás haya notado que algo que *no* hacen las definiciones TYPE anteriores es especificar los valores reales que conforman los tipos en cuestión. Esa función la realizan las *¡ ciones* asociadas al tipo (que en la sintaxis anterior se indican mediante puntos suspensivos "..."), las cuales explicaremos en el capítulo 8.

Por supuesto, también debe ser posible deshacerse de un tipo si ya no tenemos un uso adicional para él.

```
DROP TYPE <nombre de tipo> ;
```

El <*nombre de tipo*> debe identificar un tipo definido por el usuario, no uno integrado. La j operación ocasiona que la entrada del catálogo que describe al tipo sea eliminada, lo que sig-J nifica que el tipo en cuestión ya no es conocido por el sistema. *Nota:* Por razones de simplicidad, asumimos que DROP TYPE fallará si el tipo en cuestión está en uso en cualquier otra parte; en particular, si algún atributo de alguna relación, en alguna parte, está definido sobre él.

Representaciones posibles

Con el fin de ilustrar la importancia del concepto de "representación posible", consideramos un ejemplo un poco más complicado:

```
TYPE PUNTO /* puntos geométricos */
     POSSREP CARTESIANO ( X RATIONAL, Y RATIONAL )
     POSSREP POLAR ( R RATIONAL, THETA RATIONAL ) ;
```

Primero, observe que el tipo PUNTO tiene dos representaciones posibles distintas, CARTESIANO y POLAR, lo que refleja el hecho de que los puntos geométricos pueden en efecto ser representados mediante coordenadas cartesianas o polares. (Por supuesto, la representación *física* en el sistema particular en uso podría ser con coordenadas cartesianas o coordenadas polares o algo enteramente distinto). Cada representación posible tiene a su vez dos componentes. Observe en particular que este ejemplo difiere de los anteriores en que, en esta ocasión, a las representaciones posibles y a sus componentes se les han dado nombres explícitos.

Toda declaración de representación posible provoca la definición automática* de los siguientes operadores, que son más o menos fáciles de entender:

- Un operador **selector** (con el mismo nombre que la representación posible) que permite al usuario especificar o seleccionar un valor del tipo en cuestión, suministrando un valor para cada componente de la representación posible;
- Un conjunto de operadores **THE_** (uno para cada componente de la representación posible) que permiten al usuario acceder a los valores de los componentes correspondientes a la representación posible del tipo en cuestión.

Por ejemplo, aquí tenemos algunas muestras de invocaciones al selector y al operador THE_ para el tipo PUNTO:

```
CARTESIANO ( 5.0, 2.5 )
/* denota el punto con x = 5.0 , y = 2.5 */

CARTESIANO ( XXX, YYY )
/* denota el punto con x = XXX , y = YYY — */
/• XXX y YYY son variables de tipo RATIONAL */

POLAR ( 2.7, 1.0 )
/* denota el punto con r = 2.7 , theta = 1.0 */

THE_X ( P )
/* regresa la coordenada x del punto P */
/* — P es una variable de tipo PUNTO */

THE_R ( P )
/* regresa la coordenada r del punto P */
```

Nota: Como puede ver, los selectores (o más bien, las invocaciones a selectores) son una generalización del concepto de *literal* ya conocido.

Para ver cómo funcionaría lo anterior en la práctica, suponga que la representación física de los puntos está de hecho en coordenadas cartesianas (aunque en general no hay necesidad de que una representación física sea idéntica a cualquiera de las representaciones posibles especi-

---

* Aquí, por "automático" queremos decir que (a) cualquiera que sea la agencia —probablemente el sistema, probablemente algún usuario humano—, es la responsable de definir la representación posible en cuestión, también es responsable de definir los operadores correspondientes; y queremos decir que (b) hasta que dichos operadores hayan sido definidos, no está terminado el proceso de definición de esa representación posible.

ficadas). Entonces, el sistema proporcionará ciertos operadores altamente protegidos —denominados desde este momento como *seudocódigo en cursivas*— que expongan efectivamente la representación física y el *definidor del tipo* usará entonces dichos operadores para implementar los selectores CARTESIANO y POLAR necesarios.* Por ejemplo:

```
OPERATOR CARTESIANO ( X RATIONAL, Y RATIONAL )
                                    RETURNS ( PUNTO ) ¡
   BEGIN ;
      VAR P PUNTO ;
componente X de la representación física de P := X ;
componente Y de la representación física de P := Y ; RETURN
( P ) ; END ; END OPERATOR ;

OPERATOR POLAR ( R RATIONAL, THETA RATIONAL )
                                   RETURNS ( PUNTO ) ;
   RETURN ( CARTESIANO ( R * COS ( THETA ),
                         R * SIN  ( THETA ) ) ) J
END OPERATOR ;
```

Observe que la definición de POLAR hace uso del selector CARTESIANO, así como de **loa** operadores (presuntamente integrados) SIN y COS. De manera alternativa, la definición **del** POLAR podría expresarse directamente en términos de los operadores protegidos, como sigue: 1

```
OPERATOR POLAR ( R RATIONAL, THETA RATIONAL )
                                   RETURNS ( PUNTO ) ;
   BEGIN ;
      VAR P PUNTO ;
      componente X de la representación física de P
                                          := R * COS ( THETA ) ;
      componente Y de la representación física de P
                                    := R * SIN ( THETA ) ; RETURN (
P ) ; END ; END OPERATOR ;
```

El definidor de tipos también usará esos operadores protegidos para implementar los (radores THE_ necesarios, por lo tanto:

```
OPERATOR THE_X ( P PUNTO ) RETURNS ( RATIONAL ) ;
   RETURN ( componente X de la representación física de P ) ;
END OPERATOR ;

OPERATOR THE_Y ( P PUNTO ) RETURNS ( RATIONAL ) ;
   RETURN ( componente Y de la representación física de P ) ;
END OPERATOR ;

OPERATOR THE_R  ( P PUNTO ) RETURNS  ( RATIONAL ) ;
   RETURN  ( SORT  ( THE_X  ( P ) **2 + THE_Y ( P ) **2.) ) ;
END OPERATOR  ;

OPERATOR THE_THETA  (  P PUNTO )  RETURNS  ( RATIONAL ) ;
RETURN  ( ARCTAN  ( THE_Y  ( P ) /   THE_X  ( P ) ) ) ; END
OPERATOR  ;
```

* Obviamente, el definidor de tipos es —de hecho, debe ser— una excepción a la regla general de que los usuarios no estén al tanto de las representaciones físicas.

Observe que la definición de THE_R y THE_THETA hacen uso de THE_X y THE_Y, así como de los operadores (presuntamente integrados) SQRT y ARCTAN. Como alternativa, THE_R y THE_THETA podrían ser definidos directamente en términos de los operadores protegidos (dejamos los detalles como un ejercicio).

Hasta aquí el ejemplo del PUNTO. Sin embargo, es importante entender que todos los conceptos explicados se aplican también a tipos más sencillos; por ejemplo, el tipo CANT. He aquí algunos ejemplos de invocaciones de selector para ese tipo:

```
CANT ( 100 )

CANT ( C )

CANT ( ( C1 - C2 ) * 2 )
```

Y aquí algunas muestras de las invocaciones al operador THE_:

```
THE_CANT ( C )

THE_CANT ( ( C1 - C2 ) * 2 )
```

Observe en particular que debido a que los valores siempre tienen un tipo, es estrictamente incorrecto decir (por ejemplo) que la cantidad de un cierto envío es 100. Una cantidad es un valor de tipo CANT, ¡no un valor de tipo INTEGER! Por lo tanto, para el envío en cuestión, sería más propio decir que la cantidad es CANT(1OO), no sólo 100. Sin embargo, en contextos informales a menudo no nos molestamos en ser tan precisos y usamos (por ejemplo) 100, como una abreviatura cómoda de CANT(100).* En particular, hemos usado estas abreviaturas en la figura 3.8 (la base de datos de proveedores y partes) y en la figura 4.5 (la base de datos de proveedores, partes y proyectos).

Damos un último ejemplo de definición de tipo, SEGLIN (segmentos de línea):

```
TYPE SEGLIN POSSREP ( INICIO PUNTO, FIN PUNTO ) ;
```

Por supuesto, una determinada representación posible puede estar definida en términos de tipos *definidos por el usuario,* como aquí, no sólo en términos de tipos definidos por el sistema, como en todos nuestros ejemplos anteriores.

Definición de operadores

Ahora veamos más de cerca el asunto de la definición de operadores. He aquí algunos ejemplos. El primero es un operador definido por el usuario para el tipo integrado RATIONAL:

```
OPERATOR ABS ( Z RATIONAL ) RETURNS ( RATIONAL ) ¡
    RETURN ( CASE
WHEN Z > 0.0 THEN +Z / WHEN Z < 0.0
THEN  -Z END CASE ) ; END OPERATOR ;
```

---

* También hacemos lo mismo en contextos de SQL, pero por una razón distinta; específicamente debido a que SQL no soporta (todavía) los tipos definidos por el usuario. Vea el apéndice B.

El operador ABS ("valor absoluto") se define en términos de sólo un parámetro, Z, de tipo RATIONAL, y regresa un resultado del mismo tipo. (En otras palabras, una invocación a ABS —por ejemplo, ABS (AMT1 - AMT2)— es una expresión de tipo RATIONAL).

El siguiente ejemplo, DIST ("distancia entre"), involucra algunos tipos definidos por el usuario:

```
                OPERATOR DIST ( P1 PUNTO, P2 PUNTO ) RETURNS ( LONGITUD )
RETURN (          WITH ( P1 :     X1
                  THE_X ( P2      X2
                  THE_Y ( P1      Y1
                  THE_Y ( P2  AS  Y2
          LONGITUD ( SQRT         X1 X2  )  *
                                  Y1 Y2  )  *
          .
END
```

Aquí damos por hecho que LONGITUD es un tipo definido por el usuario con una representación posible RATIONAL. Observe el uso de la cláusula WITH para introducir nombres abreviados para ciertas expresiones.

El siguiente ejemplo presenta al operador de comparación "=" requerido para el tipo PUNTO:

```
OPERATOR EQ ( P1 PUNTO, P2 PUNTO ) RETURNS (  BOOLEAN )
   RETURN ( THE_X ( P1 ) = THE_X (  P2 ) AND
            THE_Y ( P1 ) = THE_Y ( P2 ) ) ;
END OPERATOR ;
```

Observe que la expresión dentro de la instrucción RETURN hace uso del operador *integrado* "=" del tipo RATIONAL. Por razones de simplicidad a partir de este momento daremos por hecho que la notación infija normal "=" puede ser usada para el operador de igualdad (para todos los tipos, incluyendo en particular el tipo PUNTO); omitimos la consideración de cómo podría es- j pecificarse esta notación infija en la práctica, ya que básicamente es sólo un asunto de sintaxis. Aquí está el operador "<" del tipo CANT:

```
OPERATOR LT ( C1 CANT, C2 CANT ) RETURNS ( BOOLEAN ) ;
   RETURN (  THE_CANT ( C1 ) < THE_CANT ( C2 ) ) ; END
OPERATOR ;
```

En este caso, la expresión dentro de la instrucción RETURN utiliza el operador integrado "<" del j tipo INTEGER. Una vez más, a partir de este momento daremos por hecho que la notación infija normal puede ser usada para este operador; es decir, para todos los "tipos ordenados", no sólo para el tipo CANT. (De hecho, por definición, un **tipo ordenado** es aquel al que se aplica "<". Un ejemplo sencillo de un tipo que definitivamente no es ordenado en este sentido es PUNTO.)

Por último, presento un ejemplo sobre la definición de un operador *de actualización* (todos **los** ejemplos anteriores fueron de operadores *sólo de lectura).** Como puede ver, la definición involucra una especificación UPDATES en lugar de una especificación RETURNS; los operadores de actualización no regresan un valor y deben ser invocados mediante llamadas explícitas (CALL) **[3.3].**

```
                                                          P ) ;
OPERATOR  REFLEJO    ( P PUNTO ) UPDATES
   BEGIN ;
      THE_X (  P )    :=  - THE _X  : p
      THE_Y (  P )    :=  - THE _Y  : p
      RETURN  ;
   END ;
END OPERATOR
```

* A los operadores sólo de lectura y de actualización también se les conoce como *observadores y muta-dores,* respectivamente, en especial en los sistemas de objetos (vea la parte VI de este libro).

El operador REFLEJO mueve efectivamente el punto con coordenadas cartesianas *(x,y)* a su posición inversa *(-x,-y);* esto lo hace actualizando apropiadamente su argumento de punto. Observe en este ejemplo el uso de las **seudovariables** THE_. Una seudovariable THE_ es una invocación al operador THE en una posición de destino (en particular, en la parte izquierda de una asignación). Esta invocación en realidad *designa* —más que regresar solamente el valor de— el componente especificado de (la representación posible aplicable de) su argumento. Por ejemplo, dentro de la definición de REFLEJO, la asignación

```
THE_X ( P )     :=     ...    ;
```

en realidad asigna un valor al componente X de (la representación cartesiana posible de) la variable argumento correspondiente al parámetro P. Por supuesto, todo argumento que va a ser actualizado por un operador de actualización, en particular mediante la asignación de la seudovariable THE_, debe indicarse de manera especí'fica como una variable, no como una expresión más general.

Las seudovariables pueden estar anidadas como sigue:

```
VAR SL SEGLIN  ;
THE_X ( THE_INICIO ( SL )                6.5
)
```

Por último, debe ser desde luego posible deshacerse de un operador si ya no tenemos un uso posterior para él. Por ejemplo:

```
DROP OPERATOR REFLEJO  ;
```

El operador especificado debe ser definido por el usuario, no uno integrado.

Conversión de tipos

Una vez más, considere la siguiente definición de tipo:

```
TYPE V# POSSREP  ( CHAR )  ;
```

De manera predeterminada, en este caso la representación posible tiene el nombre heredado V#, de ahí que el operador selector correspondiente también tenga el mismo nombre. Por lo tanto, la siguiente es una invocación de selector válida:

```
V# (  ' V1'  )
```

(la cual regresa cierto número de proveedor). Por lo tanto, observe que el selector V# podría ser considerado, a grandes rasgos, como un operador de **conversión de tipo** que convierte cadenas de caracteres a números de proveedor. En forma similar, el selector P# podría ser considerado como un operador de conversión de cadenas de caracteres en números de parte; el selector CANT podría ser considerado como un operador de conversión que convierte enteros en cantidades, y así sucesivamente.

Ahora bien, en el capítulo 3 dimos varios ejemplos en los que se realizaba una comparación entre (por ejemplo) un número de parte y una cadena de caracteres. Por ejemplo, el ejercicio 3.4 incluyó la siguiente cláusula WHERE:

```
.. .  WHERE P# =   ' P2 '
```

En este caso, el operando de la izquierda es de tipo P# y el de la derecha es de tipo CHAR; por lo tanto, frente a esto, la comparación debe fallar dando un **error de tipo** (de hecho, un error de

tipo *en tiempo de compilación).* Sin embargo, de manera conceptual, lo que sucede es que el sistema se da cuenta de que puede usar el "operador de conversión" P# para convertir el rando ope: CHAR al tipo P#, de modo que en efecto reescriba la comparación como sigue:

```
... WHERE P# = P# ( 'P2' )
```

La comparación ahora es válida.

A la acción de invocar implícitamente a un operador de conversión (de esta manera), se le conoce como coacción; y la utilizamos mucho, aunque de manera tácita, a lo largo del capí-j tulo 3. Sin embargo, es bien sabido en la práctica que la coacción puede ocasionar errores de programa. Por este motivo, a partir de este momento, adoptaremos la postura conservadora de no permitir coacciones; los operandos deberán ser siempre del tipo apropiado, no simplemente convertibles a ese tipo. En particular, insistiremos en que:

- Los operandos para "=", "<" y ">" deben ser del mismo tipo;
- Las partes a la izquierda y a la derecha de una asignación (":=") deben ser del mismo tipo.

Por supuesto, sí permitimos la definición de lo que por lo regular se conoce como operadores "CAST" (de conversión) y su invocación explícita para convertir entre tipos, donde se necesario; por ejemplo:

```
CAST_AS_RATIONAL ( 5 )
```

Como ya señalamos, también es posible pensar en los selectores —por lo menos en aquellos c sólo toman un argumento— como en operadores de conversión explícitos (de una clase).

Conclusiones

El soporte completo para los tipos comprendidos en la presente sección tiene varias imp ciones importantes, las cuales resumimos aquí brevemente:

- La primera y más importante indica que el sistema sabrá (a) exactamente qué expresio son válidas y (b) el tipo del resultado de cada una de estas expresiones válidas.
- También significa que el conjunto total de tipos para una base de datos dada será un < junto cerrado; es decir, que el tipo del resultado de toda instrucción válida será un tipo conocido por el sistema. En particular, observe que para que las comparaciones sean ex presiones válidas, este conjunto cerrado de tipos *debe* incluir el tipo *boolean* o valor de *verdad.*
- En particular, el hecho de que el sistema conozca el tipo de resultado de toda expresión válida significa que sabe qué asignaciones y qué comparaciones son válidas.

Cerramos esta sección con una referencia adelantada importante. Hemos afirmado que los dominios son tipos de datos —definidos por el sistema o por el usuario— de una complejidad interna arbitraria, cuyos valores son manipulables exclusivamente por medio de los operadores definidos para el tipo en cuestión (y cuya representación interna está, por lo tanto, oculta para el usuario). Ahora bien, si enfocamos nuestra atención por un momento en los sistemas de obje-tos. encontramos que el concepto más fundamental de objeto, la *clase objeto,* es un tipo de datos —definido por el sistema o por el usuario— de una complejidad interna arbitraria, cuyos valores son manipulables exclusivamente por medio de los operadores definidos para el tipo en

cuestión (y cuya representación interna está, por lo tanto, oculta para el usuario)... En otras palabras, ¡los dominios y las clases objeto *son lo mismo!;* de modo que aquí tenemos la clave para hacer compatibles las dos tecnologías (relaciones y objetos). En el capítulo 25 abundaremos sobre este importante asunto.

5.3 VALORES DE RELACIÓN

Del capítulo 3 recuerde que es necesario distinguir cuidadosamente, por una parte, entre las relaciones *por sí mismas* (es decir, entre los *valores* de relación) y las *variables* de relación (es decir, las variables cuyos valores son valores de relación), por la otra. En esta sección explicaré los valores de relación (relaciones, para abreviar) y en la siguiente, las variables de relación. Entonces, antes que nada, he aquí una definición precisa del término *relación:*

■ Dado un conjunto de *n* tipos o dominios *Ti (i = 1,2,..., n),* que no son necesariamente todos distintos, *r* es una **relación** sobre esos tipos si consta de dos partes: un *encabezado* y un *cuerpo;** donde:

a. El **encabezado** es un conjunto de *n* **atributos** de la forma *Ai: Ti,* donde los *Ai* (que deben ser todos distintos) son los *nombres de atributo* de *r* y los *Ti* son los *nombres de tipo* co rrespondientes (i= 1, 2, ...,n);

b. El **cuerpo** es un conjunto de *m* **tupias** *t,* en donde *t* es a su vez un conjunto de compo nentes de la forma *Ai:vi* en la cual *vi* es un valor de tipo *Ti;* es decir, el *valor de atributo* para el atributo *Ai* de la tupia t (i = 1, 2, ..., «).

A los valores *m* y *n* se les denomina **cardinalidad y grado,** respectivamente, de la relación *r.*

Los puntos que se desprenden de esta definición son:

Por supuesto, en términos de la representación tabular de una relación, el encabezado es la fila de nombres de columna y de nombres de tipo correspondientes; el cuerpo es el conjunto de filas de datos (vea más adelante una explicación mejor). 1

2 Decimos que el atributo *Ai* es *de* tipo *Ti* o (en ocasiones) que está *definido* sobre el tipo *Ti.* Observe que cualquier número de atributos —en la misma relación, en relaciones distintas, o en ambos casos— puede ser del mismo tipo (para comprender mejor este punto, vea la figura 3.8 del capítulo 3; además de las figuras 4.5 y 4.6 del capítulo 4).

3. En un momento dado, existirán comúnmente valores de un tipo determinado que no aparez can actualmente en la base de datos como un valor de alguno de los atributos existentes de ese tipo. Por ejemplo, P8 podría ser un número de parte válido aunque no exista una "parte P8" en la figura 3.8.

4. Como establecí en la definición, al valor *n* —el número de atributos en la relación— se le llama **grado** (en ocasiones, **aridad).** Se dice que una relación de grado uno es *unaría,* una relación de grado dos *binaria,* una de grado tres *ternaria, ...y* una relación de grado *n n-aria.*

---

* En la literatura también se hace referencia al encabezado como un esquema (de relación) o, en ocasiones, sólo como un esquema. También se le conoce como la intensión (nótese la ortografía), en cuyo caso al cuerpo se le denomina la extensión.

En general, el modelo relacional se ocupa entonces de las relaciones ***n-arias*** para un entero *n* cualquiera no negativo.

5. El término ***n-tupla*** se usa en ocasiones en lugar de ***tupia*** (y así hablamos de, por ejemplo, 4-tuplas, 5-tuplas, etcétera). Sin embargo, es común omitir el prefijo "n-".
6. Decir que la relación *r* tiene el encabezado *H* quiere decir precisamente que la relación r es del tipo RELATION{H} (y el nombre de ese tipo es precisamente RELATION{H}). Para una mejor explicación, vea la sección 5.4.

A manera de ejemplo, examinemos la tabla de la figura 5.1 (deliberadamente no la llamaremos relación por el momento) para ver cómo se ajusta a la definición anterior.

- Primero, esa tabla tiene cuatro tipos subyacentes; para ser específicos, números de provee dor (V#), nombres (NOMBRE), valores de status (STATUS) y nombres de ciudades (CIU DAD). *Nota:* Es cierto que cuando plasmamos una relación como una tabla sobre papel a menudo no nos molestamos en mencionar los tipos subyacentes (como vimos en el capí tulo 3); pero debemos entender que, por lo menos conceptualmente, siempre están ahí.
- A continuación, la tabla tiene en realidad dos partes: una fila de nombres de columna y un conjunto de filas de datos. Consideremos primero la fila de nombres de columna:

  ```
  ( V#, PROVEEDOR, STATUS, CIUDAD )
  ```

  Lo que en realidad representa esta fila es el siguiente *conjunto de pares ordenados:*

  ```
  { V#        : V#
    PROVEEDOR : NOMBRE ,
    STATUS    : STATUS ,
    CIUDAD    : CIUDAD }
  ```

  El primer componente de cada par es un nombre de atributo, el segundo es el nombre deli tipo correspondiente. Por lo tanto, podemos acordar que la fila de encabezados de columna sí representa un *encabezado* en el sentido de la definición. *Nota:* Como ya indiqué, en la práctica es común pensar que un encabezado sólo consiste en un conjunto de nombres de atributo (es decir, a menudo se omiten los nombres de tipo), salvo cuando la precisión es en particular importante. La práctica es imprecisa pero cómoda y con frecuencia la adoptaremos en los capítulos siguientes.
- Por lo que toca al resto de la tabla, ciertamente sí consiste en un *conjunto;* es decir, un con junto de filas de datos. Concentrémonos en una sola fila de ese conjunto, digamos la fila

  ```
  ( V1, Smith, 20, Londres )
  ```

  Lo que en realidad representa esta fila es el siguiente *conjunto de pares ordenados:*

  ```
  { V#        : V# ( ' V1 ' )        ,
    PROVEEDOR : NOMBRE  ( 'Smith' ) ,
    STATUS    : 20
    CIUDAD    : 'Londres'              }
  ```

  El primer componente en cada par es un nombre de atributo, el segundo es el valor de atributo correspondiente. Ahora bien, por lo regular en contextos informales omitimos los nombres de atributo, debido a que tenemos una convención que dice que cada valor indi-

vidual de la tabla es en realidad un valor del atributo cuyo nombre aparece en la parte superior de la columna relevante; además, ese valor es de hecho un valor del tipo subyacente relevante; es decir, el tipo del atributo en cuestión. Por ejemplo, el valor V1 —de manera más adecuada, V#('V1')— es un valor del atributo V# y es de hecho un valor del tipo relevante; es decir, el tipo "números de proveedor" (al que también se le conoce como V#). De modo que podemos acordar que cada fila representa de hecho a una *tupia*, en el sentido de la definición.

De todo lo anterior podemos acordar que la tabla de la figura 5.1 en realidad puede considerarse como una imagen de una relación, en el sentido de la definición —*siempre y cuando* podamos acordar cómo "leer" dicha imagen (es decir, *siempre y cuando* podamos acordar ciertas **reglas de interpretación** para dichas imágenes). En otras palabras, tenemos que acordar que sí existe" algunos tipos subyacentes; que cada columna sí corresponde exactamente a uno de esos tipos; que cada fila sí representa una tupia; que cada valor de atributo sí es un valor del tipo relevante; y así sucesivamente. Si podemos estar de acuerdo en todas estas "reglas de interpretación", entonces —*y sólo entonces*— podemos acordar que una "tabla" es una imagen razonable de una relación.

Entonces, ahora podemos ver que una tabla y una relación no son exactamente lo mismo (aunque en capítulos anteriores hayamos pretendido que lo eran). Más bien, una **relación** es lo que la definición dice que es (es decir, una clase de objeto más bien abstracto) y una **tabla** es una imagen concreta (generalmente sobre papel) de dicho objeto abstracto. De nuevo, una relación y una tabla no son exactamente lo mismo. Por supuesto, son muy parecidas... y por lo menos en contextos informales es usual y perfectamente aceptable decir que sí lo son. Pero cuando intentamos ser más precisos (y, desde luego, en este momento *estamos* intentando serlo), entonces tenemos que reconocer que los dos conceptos no son exactamente idénticos.

Nota: Si tiene problemas con la idea de que existen algunas diferencias entre una relación y una tabla, tal vez le ayude lo siguiente. Primero, es innegable que una gran ventaja del modelo relacional es que su objeto abstracto básico, la relación, tiene una representación muy sencilla sobre el papel; es esa representación la que hace que los sistemas relacionales sean fáciles de usar y entender, y facilita el razonamiento sobre la forma en que los sistemas relacionales se comportan. Sin embargo, también es desafortunado el hecho de que la representación tabular *sugiere algunas cosas que no son ciertas.* Por ejemplo, claramente sugiere que las filas de la tabla —es decir, las tupias de la relación— están en un cierto orden de arriba hacia abajo, aunque en realidad no lo están (vea la siguiente subsección).

Propiedades de las relaciones

Las relaciones poseen ciertas propiedades, todas ellas consecuencias inmediatas de la definición de *relación* dada en la subsección anterior, y todas ellas muy importantes. Primero enunciaremos brevemente las propiedades y luego las explicaremos en detalle. Las propiedades son como sigue. Dentro de cualquier relación dada:

- No existen tupias duplicadas;
- Las tupias están en desorden, de arriba hacia abajo;
- Los atributos están en desorden, de izquierda a derecha;

■ Cada tupia contiene exactamente un valor para cada atributo.

1. *No existen tupias duplicadas*

Esta propiedad surge del hecho de que el cuerpo de la relación es un conjunto matemático (de tupias); los conjuntos en matemáticas no incluyen elementos duplicados.

Nota: De hecho, debe ser obvio que el concepto de "tupias duplicadas" no tiene sentido. Considere una relación con los atributos V# y CIUDAD (que significan que "el proveedor V# está ubicado en la ciudad CIUDAD"). Suponga que la relación contiene una tupia que muestra que es un "hecho verdadero" que el proveedor V1 está ubicado en Londres. Entonces, si la relación contuviera un duplicado de esa tupia (si eso fuese posible), simplemente nos estaría diciendo, por segunda vez, el mismo "hecho verdadero". Pero si algo es cierto, ¡decirlo dos veces no lo hace *más* cierto!

A propósito, esta primera propiedad sirve para ilustrar el punto de que (en general) una relación y una tabla no son lo mismo; ya que (en general) una tabla puede contener filas duplicadas —en ausencia de cualquier disciplina que evite tal cosa—, mientras que una relación *no puede* contener una tupia duplicada. (Ya que si una "relación" contiene tupias duplicadas, entonces, por definición, *¡no es una relación!)* Es muy desafortunado el caso de que SQL permita que las tablas contengan filas duplicadas. Éste no es el lugar apropiado para abordar todas las razones por las cuales tales duplicidades son un error (para una explicación amplia, vea las referencias [5.3] y [5.6]); para los fines actuales, es suficiente señalar que el modelo relacionala) reconoce duplicados, y por lo tanto, en este libro nos aseguraremos que nunca ocurran. (Esta observación se aplica principalmente a nuestras explicaciones de SQL. Por supuesto, en lo que respecta al modelo relacional en sí mismo, no es necesario tener un cuidado especial.)

2. *Las tupias están en desorden, de arriba hacia abajo*

Esta propiedad también surge del hecho de que el cuerpo de la relación es un conjunto matemático; en matemáticas, los conjuntos no están ordenados. Por ejemplo, en la figura 5.1 las tupias también podrían haber sido mostradas en secuencia inversa, y seguiría siendo la misma relación. Por lo tanto, no hay algo que se llame "la quinta tupia" o "la 97a tupia" o "la primen tupia" de una relación y tampoco hay algo como "la siguiente tupia"; en otras palabras, no existe el concepto de direccionamiento posicional y no existe el concepto de "la que sigue". La referencia [5.6], mencionada anteriormente con relacion a la propiedad de que "no hay tupias duplicadas", muestra por qué también es tan importante la propiedad de "no ordenamiento de tupias" (de hecho, ambas propiedades están interrelacionadas).

Nota: Es cierto que se requiere algún concepto como el de "la que sigue" en una interíaz entre la base de datos y un lenguaje anfitrión como C o COBOL (vea la explicación de cursores de SQL y la cláusula ORDER BY en el capítulo 4). Pero es el lenguaje anfitrión y no el modelo relacional el que impone ese requerimiento. En efecto, el lenguaje anfitrión requiere que conjuntos *sin orden* se conviertan en listas o arreglos (de tupias) *ordenados,* de modo que las operaciones como "ac-ceder a la siguiente tupia" puedan tener significado. Observe además que dichas características sólo forman parte de la interfaz del programa de aplicación; no están expuestas a los usuarios finales.

Como mencioné anteriormente en esta sección, esta segunda propiedad sirve también para ilustrar la idea de que una relación y una tabla no son lo mismo, ya que las filas de una tabla obviamente tienen un ordenamiento de arriba hacia abajo, mientras que las tupias de una relación no lo tienen.

3. *Los atributos están en desorden, de izquierda a derecha*

Esta propiedad surge del hecho de que el encabezado de una relación también es un conjunto (de atributos). Por ejemplo, en la figura 5.1 los atributos también podrían haber sido mostrados en el orden (por decir) PROVEEDOR, CIUDAD, STATUS, V#, y seguiría siendo la misma relación (por lo menos en lo que respecta al modelo relational).* Por lo tanto, no existe algo como "el primer atributo" o "el segundo atributo" (etcétera) y tampoco hay "el siguiente atributo" (de nuevo, no existe el concepto de "el siguiente"); en otras palabras, siempre se hace referencia a los atributos por nombre, nunca por posición. Como resultado, se reduce el alcance de los errores y la programación confusa. Por ejemplo, no hay (o no debe haber) una forma de hacer que el sistema "se extienda" de algún modo de un atributo a otro. Esta situación contrasta con la que se encuentra en muchos sistemas de programación, donde a menudo es posible explotar de muy diversas formas la adyacencia física de elementos lógicamente discretos, de manera deliberada o no.

Observe que esta cuestión del ordenamiento de los atributos es otra área más en donde la representación concreta de una relación como una tabla, sugiere algo que no es cierto: las columnas de una tabla tienen obviamente un ordenamiento de izquierda a derecha, pero los atributos de una relación no lo tienen.

4. *Cada tupia contiene exactamente un valor para cada atributo*

Esta propiedad surge inmediatamente de la definición de una tupia: una tupia es un conjunto de *n* componentes o pares ordenados de la forma *Ai:vi (i*= 1,2,..., *n).* Se dice que una relación que satisface esta cuarta propiedad está **normalizada** o, lo que es equivalente, está en la **primera forma normal.**

Nota: Esta propiedad en particular podría parecer muy obvia y de hecho lo es; en especial debido a que (como tal vez haya notado) *todas* las relaciones están normalizadas de acuerdo con la definición. Sin embargo, esta propiedad sí tiene algunas consecuencias importantes. Vea (a) la nota a la referencia [5.10], (b) el capítulo 18 sobre información faltante y (c) la subsección siguiente.

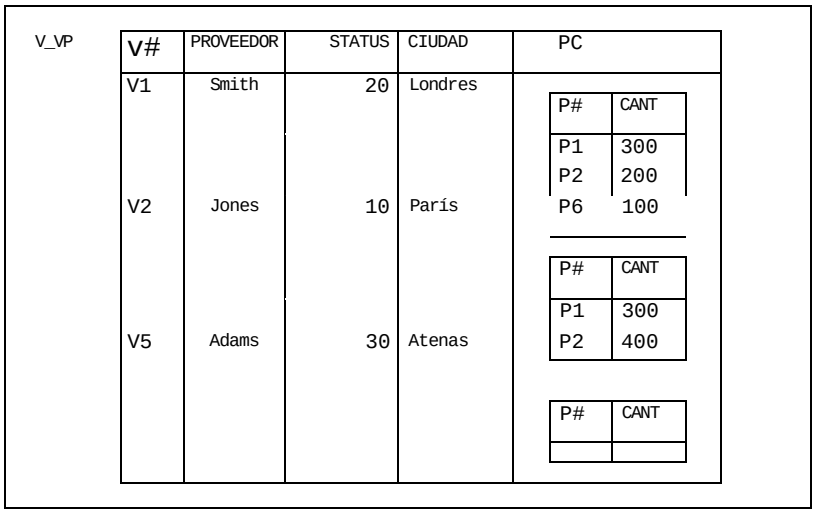
Atributos con valor de relación

En la sección 5.2, dijimos que los valores que conforman un tipo pueden ser **de cualquier clase.** Por lo tanto, podemos tener un tipo en particular cuyos valores sean *relaciones y* por lo tanto tener relaciones con atributos cuyos valores sean a su vez relaciones. En otras palabras, podemos tener relaciones que tengan otras relaciones anidadas dentro de sí mismas. La figura 5.3 muestra un ejemplo: la relación V_VP; el atributo PC de esa relación tiene el valor de una relación. Observe en particular que el conjunto vacío de partes que suministra el proveedor V5 está representado por un conjunto vacío (para ser más precisos, por una relación vacía).

Aquí, el motivo principal para mencionar explícitamente la posibilidad de que los atributos tengan valores de relación es que, históricamente, la mayoría de las publicaciones de bases de datos relacionales —incluidas las primeras ediciones de este libro— han afirmado que dicha

---

* Las relaciones matemáticas, a diferencia de sus contrapartes en el modelo relacional, sí tienen un orden de izquierda a derecha para sus atributos.



V_VP

V#	PROVEEDOR	STATUS	CIUDAD	PC
V1	Smith	20	Londres	P# CANT: P1 300; P2 200
V2	Jones	10	París	P6 100
V5	Adams	30	Atenas	P# CANT: P1 300; P2 400
				P# CANT:

Figura 5.3 Una relación con un atributo que tiene valor de relación.

posibilidad no es válida (y la mayoría aún lo afirman). Por ejemplo, el siguiente es un extracto ligeramente modificado de la edición anterior de este libro:

> Observe que *todos los valores de columnas son atómicos...* Es decir, en cada posición de fila y columna de una tabla siempre hay exactamente un valor de datos, nunca un grupo de varios valores. Por lo tanto, tenemos (por ejemplo) en la tabla EMP

DEPTO#	EMP#
D1 D1	E1 E2

en lugar de

DEPT0#	EMP#
D1	E1 , E2

> En la segunda versión de esta tabla, la columna EMP# es un ejemplo de lo que por lo regular se denomina **grupo de repetición.** Un grupo de repetición es una columna... que contiene varios valores en cada fila (en general, diferente número de valores en diferentes filas). *Las bases de datos relacionales no permiten los grupos de repetición;* la segunda versión de la tabla anterior no sería permitida en un sistema relacional.

Y más adelante, en el mismo libro:

> [Los dominios] contienen sólo valores atómicos... [Por lo tanto,] *las relaciones no contienen grupos de repetición.* Se dice que una relación que satisface esta condición está normalizada o, lo que es equivalente, está en la primera forma normal... En el contexto del modelo relacional, el término *relación* siempre se utiliza para indicar una relación normalizada.

Sin embargo, estas observaciones no son correctas (por lo menos en su totalidad). Surgieron de una mala interpretación de mi parte sobre la verdadera naturaleza de los tipos y los predicados. Por razones que explicaré en el capítulo 11 (sección 11.6), no es probable que este error haya ocasionado errores serios en la práctica; sin embargo, sigo ofreciendo mis disculpas a cualquiera a quien esto haya causado confusión. Por lo menos la edición anterior estuvo correcta cuando dijo que en el modelo relacional las relaciones siempre estaban normalizadas. Consulte, una vez más, el capítulo 11 para una mayor explicación.

Las relaciones y su interpretación

Concluimos esta sección con un recordatorio de que —como expliqué en el capítulo 3, sección 3.4— (a) el encabezado de cualquier relación dada pueda ser considerado como un *predicado* y (b) las tupias de esa relación puedan ser consideradas como *proposiciones verdaderas* obtenidas del predicado mediante la sustitución de valores del tipo apropiado por los *parámetros o indicadores de posición* de ese predicado ("lo que crea un ejemplar del predicado"). De hecho, la **Suposición del Mundo Cerrado** (también conocida como la **Interpretación del Mundo Cerrado)** dice que si una tupia válida —es decir, una que se apegue al encabezado de la relación— *no* aparece en el cuerpo de la relación, entonces podemos pensar que la proposición correspondiente es *falsa.* Esto es, el cuerpo de la relación contiene *únicamente* aquellas tupias que correspondan a proposiciones verdaderas.

5.4 VARIABLES DE RELACIÓN

Del capítulo 3, recuerde que las variables de relación —o *variables relacionales* o *varrels,* para abreviar— se presentan en dos variedades: las varrels base y las vistas (también denominadas varrels auténticas y virtuales, respectivamente). En esta sección consideraremos únicamente a las varrels base; explicaré las vistas en el capítulo 9.

Definición de varrel base

Aquí tenemos la sintaxis para definir una varrel base:

```
VAR <nombre de varrel> BASE <tipo de relación>
                          <lista de definición de claves candidatas> [
                        <lista de definición de claves externas> J
```

El <*tipo de relación*> toma la forma

```
RELATION {   <lista de atributos separados con comas>   }
```

en donde cada <*atributo*> es a su vez un par ordenado de la forma

```
<nombre de atributo> <nombre de tipo>
```

Con respecto a la <*lista de definición de claves candidatas*> y la opcional <*lista de definición de claves externas*>, vea más adelante.

Nota: El término *lista con comas* lo definí en el capítulo 4 (sección 4.6). El término *lisa* lo defino de manera análoga, por lo tanto: Sea <*xyz*> un elemento para denotar una categoría sintáctica cualquiera (es decir, todo lo que aparezca a la izquierda de una regla de producción BNF).. Entonces la expresión <*lista de xyz*> denota una secuencia de cero o más elementos <.rv;>.a la cual un par de <*xyz*> adyacentes está separado por uno o más espacios en blanco.

Aquí tenemos, a manera de ejemplo, las definiciones de varrel base para la base de datos de proveedores y partes (retomada de la figura 3.9):

```
VAR V BASE RELATION
   { V#          V#,
     PROVEEDOR   NOMBRE,
     STATUS      INTEGER,
     CIUDAD      CHAR }
   PRIMARY  KEY { V# } ;

VAR P BASE RELATION

   { P#          P#,
     PARTE       NOMBRE,
     COLOR       COLOR,
     PESO        PESO,
     CIUDAD      CHAR }
   PRIMARY  KEY { P# } ;

VAR VP      RELATION
BASE
   { V#       V#,
     P#       P#,
     CANT     CANT }
   PRIMARY  KEY { V#, P# }
   FOREIGN  KEY { V# } REFERENCES
                        FOREIGN KEY { P# } REFERENCES P ;
```

Explicación:

1. Los **tipos** (de relación) de estas tres varrels base son como sigue:

   ```
   RELATION { V# V#, PROVEEDOR NOMBRE, STATUS INTEGER, CIUDAD CHAR }

   RELATION { P# P#, PARTE NOMBRE, COLOR COLOR,
                                   PESO PESO, CIUDAD CHAR }

   RELATION { V# V#, P# P#, CANT CANT }
   ```

   Observe que cada uno de estos tipos de relación es de hecho un tipo y puede ser usado todas las formas normales; en particular, como el tipo para algún atributo en alguna relación. (De hecho, RELATION es un **generador de tipos** que nos permite definir 6 tipos de relación específicos como queramos, en forma muy similar a como "ARRAY" es un generador de tipos en los lenguajes de programación convencionales.)

2. Todos los valores de relación posibles de cualquier varrel dada (sea base u otra), mismo tipo de relación (es decir, el tipo de relación especificado directa o indirectamen-

te en la definición de la varrel) y por lo tanto, tienen el mismo encabezado. En particular, el valor inicial de cualquier varrel base dada es una relación vacía del tipo correspondiente.

3. Todos los términos *encabezado, cuerpo, atributo, tupia, cardinalidady grado,* ya definidos para los valores de relación, se interpretan en la forma obvia para aplicarse también a las varrels (es decir, a todas las varrels, no sólo a las base).
4. Definir una nueva varrel base ocasiona que el sistema cree entradas en el catálogo para des cribirla.
5. En el capítulo 8, explico en detalle las definiciones de claves candidatas. Pero antes de lle gar a ese punto, simplemente daremos por hecho que cada definición de varrel base com prende exactamente una de las definiciones de la siguiente forma en particular:

```
PRIMARY KEY { <lista de nombres de atributo separados con comas> }
```

6. En el capítulo 8, también explicaré las definiciones de clave externa.

Por último, presento la sintaxis para eliminar una varrel base:

```
DROP VAR <nombre de varrel>  ;
```

Esta operación asigna el valor de la varrel base especificada a una relación vacía (es decir, hablando en términos generales, elimina todas las tupias de la varrel) y luego elimina todas las entradas del catálogo para esa varrel. La varrel ya no será conocida para el sistema. *Nota:* Por razones de simplicidad, damos por hecho que la eliminación (DROP) fracasará si la varrel todavía está en uso en alguna parte; por ejemplo, si se hace mención de ella en una vista. Vea el capítulo 9 para una mejor explicación.

Actualización de varrels

El modelo relacional incluye una operación de **asignación relacional** para asignar valores a varrels (en particular, varrels base); es decir, para *actualizarlas.* La siguiente es la sintaxis en **Tutorial D:**

```
<nombre de varrel>  := <expresión relacional>  ;
```

La <*expresión relacional*> es evaluada y la relación que resulta de esa evaluación es asignada a la varrel identificada por <*nombre de varrel*>, reemplazando el valor anterior de dicha variable. Por supuesto, la varrel y la relación en cuestión deben ser del mismo tipo (es decir, tener el mismo encabezado).

A manera de ejemplo, suponga que estamos dando otra varrel R del mismo tipo que la variable de proveedores V:

```
VAR R BASE RELATION
    { V# V#, PROVEEDOR NOMBRE, STATUS INTEGER, CIUDAD CHAR } ... ;
```

Aquí tenemos entonces algunas asignaciones relacionales válidas:

```
■ R := V ;
■ R := V WHERE CIUDAD = 'Londres' ;
■ R := V MINUS ( V WHERE CIUDAD = 'París' ) ;
```

Observe que cada uno de estos ejemplos puede ser considerado como (a) que *recupera la* relación que está a la derecha y (b) que *actualiza* la varrel que está en la parte izquierda.

Suponga ahora que cambiamos el segundo y tercer ejemplos reemplazando en cada caso la varrel R de la parte izquierda por la varrel V:

```
■ V := V WHERE CIUDAD - 'Londres' ;
■ V := V MINUS ( V WHERE CIUDAD • 'París' ) ;
```

Observe que estas dos asignaciones son en efecto *actualizaciones a la varrel V;* una elimina efectivamente todos los proveedores que no están en Londres y la otra elimina efectivamente a todos los proveedores en Pans. Por conveniencia, **Tutorial D** soporta operaciones INSET. DELETE y UPDATE explícitas, pero cada una de estas operaciones se define como una ¡ viatura de una cierta asignación relacional. He aquí algunos ejemplos:*

```
■ INSERT INTO V
  RELATION { TUPLE { V# V# ( 'V6' ),
                     PROVEEDOR NOMBRE ( 'Smith' ),
                     STATUS 50,
                     CIUDAD 'Roma' } } ¡
```

Asignación equivalente:

```
V := V UNION
       RELATION { TUPLE { V# V# ( 'V6' ),
                     PROVEEDOR NOMBRE ( 'Smith' ),
                     STATUS 50,
                     CIUDAD 'Roma' } } ;
```

A propósito, observe que esta asignación tendrá éxito si la tupla especificada para el proveedor V6 ya aparece en la varrel V. En la práctica, podríamos desear ampliar la asignación análoga de INSERT para asegurar que éste no sea el caso; sin embargo, por razones de simplicidad, ignoraremos aquí este detalle. Por supuesto, se aplican observaciones análogas a DELETE y UP DATE.

```
■ DELETE V WHERE CIUDAD = 'París' ;
```

Asignación equivalente:

```
V := V MINUS ( V WHERE CIUDAD = 'París' ) ;
```

---

* La expresión RELATION {...} en el ejemplo de INSERT es una *invocación al selector de relación (y en* realidad, la expresión dentro de esas llaves, TUPLE (…}, es a su vez una invocación al selector de tupla y las expresiones dentro de esas llaves son invocaciones al selector *escalar).* Vea el capítulo 6, seo 6.3 y 6.4.

```
■ UPDATE V WHERE CIUDAD = 'París',
               STATUS := 2 * STATUS,
               CIUDAD := 'Roma' ;
```

La asignación equivalente es un poco complicada en el caso de UPDATE, por lo que omitimos aquí los detalles. Vea la referencia [3.3].

Para fines de consulta, presento un resumen simplificado de la sintaxis de INSERT, DELETE y UPDATE:

```
■ INSERT INTO <nombre de varrel> <expresión relacional> ;

■ DELETE <nombre de varrel> [ WHERE <expresión lógica> ] ;

■ DELETE <nombre de varrel> [ WHERE <expresión lógica> ]
            <lista de actualizaciones de atributo separadas con comas> ;
```

A su vez, una <*actualiz.ación de atributos*> toma la forma:

<nombre de atributo> :■ <expresión>

Considero que la sintaxis de <*expresión lógica*> es más o menos clara; sin embargo, proporciono más detalles en el capítulo 6.

Cerramos esta subsección haciendo énfasis en el punto de que la asignación relacional, y por lo tanto las operaciones INSERT, UPDATE y DELETE, son operaciones **en el nivel de conjunto.** En términos generales, UPDATE (por ejemplo) actualiza un *conjunto* de tupias en la varrel de destino. De manera informal, a menudo hablamos de (por ejemplo) actualizar alguna tupia individual, pero debe entenderse claramente que:

1. En realidad hablamos de actualizar un *conjunto* de tupias, un conjunto que precisamente tiene una cardinalidad uno, y
2. ¡En ocasiones es imposible actualizar un conjunto de tupias que tiene cardinalidad uno!

Por ejemplo, suponga que la varrel de proveedores está sujeta a la restricción de integridad (vea el capítulo 8) de que los proveedores V1 y V4 deben tener el mismo status. Entonces, cualquier actualización de "tupia individual" que intente cambiar el status de uno solo de esos dos proveedores, necesariamente debe fallar. En su lugar, ambos deben ser actualizados al mismo tiempo, como aquí:

```
UPDATE V WHERE V# = V# ( 'V1' ) OR V# = V# ( 'V4' )
         STATUS := algún valor ;
```

Para continuar por un momento con la idea anterior, debemos ahora agregar que hablar (como acabamos de hacerlo) de "actualizar una tupia" o un conjunto de tupias es más bien impreciso. Las tupias, al igual que las relaciones, son *valores y no pueden* ser actualizadas (por definición, lo único que no puede usted hacer a *ningún* valor es cambiarlo). A fin de poder "actualizar tupias", necesitaríamos alguna noción de una *variable* de tupia o "vartupla"; ¡una noción que no forma parte en absoluto del modelo relacional! Por lo tanto, lo que en realidad queremos decir cuando hablamos de "actualizar la tupia *t* a t'", por ejemplo, es que estamos **reemplazando** la tupia *t* (es decir, el *valor* de la tupia *t)* por otra tupia *t'* (que es una vez más, un *valor* de tupia). Se aplican observaciones similares a frases como "actualizar el atributo *A"* (en alguna tupia). En este libro, continuaremos hablando en términos de "actualizar tupias" o

"actualizar atributos de tupias" —la práctica es conveniente— pero debemos entender que tal uso es sólo una forma de abreviar y más bien una forma imprecisa de abreviar.

5.5 PROPIEDADES DE SQL

Las instrucciones SQL más importantes para el material expuesto en las secciones anteriores son:

```
CREATE DOMAIN     CREATE TABLE       INSERT
ALTER DOMAIN      ALTER TABLE        UPDATE
DROP DOMAIN       DROP TABLE         DELETE
```

En el capítulo 4 expliqué las instrucciones INSERT, UPDATE y DELETE. A continuatión **con**-sideraremos las otras instrucciones.

Dominios

Como observamos en el capítulo 4, por desgracia los "dominios" de SQL están muy lejos de ser verdaderos dominios relacionales (es decir, tipos); de hecho, los dos conceptos están tan di* tantes, que hubiera sido preferible usar otro nombre para dicha construcción en SQL. La finalidad principal de los dominios en SQL es simplemente permitir que se le dé un nombre a un tipo integrado para que pueda ser usado como abreviatura por varias columnas en varias definiciones de tabla base. He aquí algunos ejemplos:

```
CREATE DOMAIN V# CHAR(5) ;
CREATE DOMAIN P# CHAR(6) ;

CREATE  TABLE  V  (  V#  V#,  ...  )  ;
CREATE TABLE P ( P# P#, ... ) ; CREATE
TABLE VP ( V# V#, P# P#, ... ) ;
```

Para fines de consulta, en la lista siguiente presentamos algunas diferencias ente los ver-daderos dominios y la construcción en SQL:

- Como ya indiqué, los dominios de SQL son en realidad una abreviatura sintáctica; no **son** verdaderos tipos de datos (como tales) y ciertamente no son tipos completos *definidos por el usuario.*
- Los valores en los dominios de SQL ciertamente no pueden ser "de una complejidad interna cualquiera" sino que están limitados a la complejidad —cualquiera que ésta sea— de los tipos integrados.
- Un dominio de SQL debe estar definido en términos de uno de los tipos integrados, no de otro dominio definido por el usuario.
- De hecho, un dominio de SQL debe estar definido en términos de *un solo* tipo integrad Por lo tanto, no es posible (por ejemplo) definir dominios de SQL análogos a los PUNTO y SEGLIN de la sección 5.2.
- Un dominio de SQL no puede tener más de una representación "posible". De hecho, los dominios de SQL no distinguen adecuadamente entre los tipos y las representaciones **(físicas** Por ejemplo, los dominios de SQL V# y P# están definidos en términos del tipo CHAR; por ello SQL nos permitirá realizar (por ejemplo) operaciones de concatenación nas sobre los números de proveedor y los números de parte.

- De aquí surge que no hay tipos fuertes y que hay muy poca comprobación real de tipos. Por ejemplo, dadas las definiciones de V# y P# mostradas anteriormente, la siguiente operación de SQL *no* fallará en una comprobación de tipo, aunque lógicamente debería:

```
SELECT *
FROM   VP
WHERE V# = P# ;
```

Nota: Tal vez una forma más cortés de decir lo mismo sea que SQL soporta exactamente ocho dominios relacionales verdaderos; es decir, los siguientes ocho "tipos básicos":

- números
- cadenas de caracteres
- cadenas de bits
- fechas
- horas
- marcas de tiempo
- intervalos año/mes
- intervalos día/hora

Entonces podríamos decir que se realiza la comprobación de tipos, pero sólo con base en estos ocho tipos básicos. De ahí que, por ejemplo, no sea válido un intento de comparación entre un número y una cadena de bits, pero que sea válido un intento de comparación entre dos números, aun cuando esos números tengan distintas representaciones (digamos, uno como INTEGER y el otro como FLOAT).

- SQL no permite que los usuarios definan los operadores que se apliquen a un dominio dado. En su lugar, los operadores que se aplican son precisamente aquellos operadores integra dos que se aplican para la *representación* correspondiente.

Entonces, la sintaxis de la instrucción **CREATE DOMAIN** de SQL es:

```
CREATE DOMAIN <nombre de dominio> <nombre de tipo integrado> [
              especificación predeterminada>  ] [
              <restricción(es)> ]  ;
```

Explicación:

1. Para una lista de los <*nombres de tipos integrados*> válidos, consulte el capítulo 4, sec ción 4.2.
2. La <*especificación predeterminada*> opcional especifica un valor predeterminado que se aplica a toda columna definida sobre el dominio que no tenga un valor predeterminado ex plícito propio (vea la siguiente subsección). Toma la forma "DEFAULT <*valor predeter minado*>*";* donde <*valor predeterminado*> es a su vez una literal, un nombre de operador
   integrado niládico o NULL.* *Nota:* Un operador niládico es aquel que no toma operandos (un ejemplo es CURRENT_DATE).
3. Las <*restricciones*> opcionales especifican ciertas restricciones de integridad que se apli can al dominio en cuestión. Dejamos su explicación para el capítulo 8.

---

* La explicación referente al soporte de SQL para los "nulos" la posponemos hasta el capítulo 18. Sin embargo, es inevitable hacer referencia a ello antes de llegar a ese punto.

En cualquier momento, un dominio SQL existente también puede ser *alterado* de diversas formas por medio de la instrucción **ALTER DOMAIN.** En particular, ALTER DOMAIN permite definir una nueva <*especificación predeterminada*> para un domino existente o eliminar una existente. También permite definir una restricción de integridad nueva para un dominio existente o eliminar una ya existente. Sin embargo, los detalles de estas opciones son sorprendentemente complejos y exceden el alcance de este libro; para los detalles específicos, consulte la referencia [4.19].

Por último, un dominio existente de SQL puede ser *eliminado* mediante la instrucción **DROP DOMAIN,** cuya sintaxis es:

```
DROP DOMAIN <nombre de dominio> <opción> ;
```

donde <*opción*> puede ser RESTRICT o bien, CASCADE. La idea general es como sigue:

- Si se especifica RESTRICT, la eliminación fracasará cuando haya referencias al dominio en cualquier parte;
- Si se especifica CASCADE, la eliminación tendrá éxito y tendrá un efecto de 'cascada" en .
  varias formas. Por ejemplo, las columnas que se definieron previamente sobre el dominio, ahora se considerarán como directamente definidas sobre el tipo de datos subyacente del do minio.

Una vez más, los detalles son muy complejos y los omitimos aquí. Para mayor información, vea la referencia [4.19].

Tablas base

Antes de entrar de manera específica en los detalles de las tablas base, hay un par de puntos) señalar sobre el tema de las tablas de SQL en general:

- Primero, se permite que las tablas de SQL incluyan **filas duplicadas.** Por lo tanto, no necesariamente tienen una clave primaria (o, de manera más fundamental, claves candidatas).
- Segundo, se considera que las tablas de SQL tienen un ordenamiento de izquierda a **derecha.** Por ejemplo, en la tabla V de proveedores, la columna V# podría ser la primera columna, la columna PROVEEDOR podría ser la segunda, etcétera.

Pasemos ahora específicamente a las tablas base. Las tablas base se definen por medio de la instrucción **CREATE TABLE** (por lo tanto, observe que aquí la palabra reservada TABLE se refiere específicamente a una tabla base; esto mismo es cierto para ALTER TABLE y DROP TABLE). La sintaxis es la siguiente:

```
CREATE TABLE <nombre de tabla base>
            ( <lista de elementos de la tabla base separados con comas > ) ;
```

donde cada <*elemento de la tabla baso* es una <*definición de columna*> o bien una <***restric-ción***>. Las <*restricciones*> especifican ciertas restricciones de integridad que se aplican a las tablas base en cuestión; dejamos la explicación detallada de dichas restricciones para el capítulo 8. Las <*defmiciones de columna*> (debe existir por lo menos una) toman la siguiente forma general:

```
<nombre de columna> <nombre de tipo o de dominio>  [ <especificación
predeterminada> ]
```

Aquí, el <*nombre de tipo o de dominio* es un nombre de tipo integrado o bien un nombre de dominio, y la <*especificación predeterminada*> opcional especifica un valor predeterminado para la columna, el cual sustituye a cualquier valor predeterminado especificado en el nivel del dominio (en caso de que sea aplicable). *Nota:* Una <*especificación predeterminada*> define el *valor predeterminado,* que se colocará en la columna aplicable si el usuario no proporciona un valor explícito en INSERT. Para aclarar esto, vea el capítulo 4, sección 4.6, subsección "Operaciones que no involucran cursores". Si una columna determinada no tiene un valor predeterminado explícito propio, y no hereda uno del dominio subyacente, se da por hecho de manera implícita que tiene un valor predeterminado NULL; es decir, NULL es "el valor predeterminado del valor predeterminado".

Para algunos ejemplos de CREATE TABLE, vea la figura 4.1 del capítulo 4.

A continuación, una tabla base existente puede ser *alterada* en cualquier momento mediante la instrucción **ALTER TABLE.** Las siguientes "alteraciones"son soportadas:

- Es posible agregar una nueva columna.
- Es posible definir un valor predeterminado para una columna existente (reemplazando, si lo hay, al anterior).
- Es posible eliminar el valor predeterminado de una columna existente.
- Es posible eliminar una columna existente.
- Es posible especificar una nueva restricción de integridad.
- Es posible eliminar una restricción de integridad existente.

Sólo ofrecemos un ejemplo del primer caso:

```
ALTER TABLE V ADD COLUMN DESCUENTO INTEGER DEFAULT -1 ;
```

Esta instrucción agrega la columna DESCUENTO (del tipo INTEGER) a la tabla base de proveedores. Todas las filas de esa tabla se amplían de cuatro a cinco columnas; en todos los casos, el valor inicial de la quinta columna nueva es -1.

Por último, una tabla base puede ser eliminada mediante **DROP TABLE;** la sintaxis es:

```
DROP TABLE <nombre de tabla base> <opción> ;
```

donde (al igual que en DROP DOMAIN) <*opción*> es RESTRICT o bien CASCADE. Si se especifica RESTRICT y se hace referencia a la tabla base en cualquier definición de vista o restricción de integridad, la eliminación fracasará; si se especifica CASCADE, la eliminación tendrá éxito (eliminando la tabla y todas sus filas) y se eliminarán también todas las definiciones de vista y restricciones de integridad que hagan referencia a esa tabla.

5.6 RESUMEN

En este capítulo vimos a fondo la parte **estructural** del modelo relacional. Un **dominio** es sólo un **tipo de datos** (posiblemente integrado o definido por el sistema; para ser más específicos, definido por el usuario); el dominio proporciona (a) un conjunto de **valores** (todos los valores posibles del tipo en cuestión) a partir de los cuales diversos atributos en distintas relaciones toman sus valores reales y (b) un conjunto de **operadores** (tanto de *sólo de lectura* como de *ac-*

tualización) para operar sobre valores y variables del tipo en cuestión. Los valores de un tipo dado pueden ser **de cualquier clase** (números, cadenas, fechas, horas, grabaciones de sonido. mapas, grabaciones de vídeo, puntos geométricos, etcétera). Los tipos restringen **a** las opera **ciones,** ya que se requiere que los operandos para una operación determinada sean del tipo apropiado para esa operación *(tipos fuertes).* Los tipos fuertes son una buena idea, ya que permiten capturar ciertos errores de lógica y hacerlo además en tiempo de compilación, en lugar de hacer-lo en tiempo de ejecución. Observe que, como veremos en el siguiente capítulo, los tipos fuerte tienen implicaciones importantes para las operaciones *relacionales* en particular (junta, unión. etcétera).

Los tipos pueden ser **escalares** o **no escalares.** Un tipo escalar es aquel que no tiene componentes visibles para el usuario. Los tipos *no* escalares más importantes en el modelo relacional son los tipos *de relación* (vea el siguiente párrafo), los cuales se definen mediante el generador **de tipos RELATION.** Distinguimos cuidadosamente entre un tipo y su **representación** física (los tipos son un aspecto del *modelo,* las representaciones físicas son un aspecto de la *implementación).* Sin embargo, requerimos que cada tipo escalar tenga declarada por lo menos una representación **posible.** Cada una de estas representaciones posibles ocasiona la definición automática (a) de un operador **selector** y (b) para cada componente de esa representación posible,un operador **THE_** (incluyendo una **seudovariable** THE_). Manejamos **conversiones** explícitasde tipo, pero no **coacciones** implícitas de tipo. Manejamos también la definición de cualquier cantidad de operadores adicionales para los tipos escalares y requerimos que se defina un operador de **igualdad** para cada tipo (¡con semántica prescrita!).

Pasemos ahora a las relaciones. Distinguimos entre **valores** de relación ("relaciones"); **variables** de relación ("varrels"). Una **relación** tiene dos partes: un encabezado y un cuerpo; el encabezado es un conjunto de **atributos,** mientras que el cuerpo es un conjunto de tupias que se apegan al encabezado. Al número de atributos del encabezado se le denomina grado y al número de tupias del cuerpo se le conoce como **cardinalidad.** Una relación tiene un tipo de **relación;** para ser específicos, el tipo RELATION{//}, donde *H* es el encabezado aplicable.Es posible ver una relación como una **tabla,** con las columnas representando a los atributo filas a las tupias, aunque esta representación es sólo aproximada. Además, todas las relaciones satisfacen cuatro propiedades muy importantes:

- No contienen tupias duplicadas.
- No hay un ordenamiento para las tupias, de arriba hacia abajo.
- No hay un ordenamiento para los atributos, de izquierda a derecha.
- Cada tupia contiene exactamente un valor para cada atributo (es decir, todas las relaciones están **normalizadas** o, lo que es lo mismo, están en la **primera forma normal).**

Como vimos en el capítulo 3, es posible considerar al encabezado de una relación como un **predicado** y a las tupias de esa relación como **proposiciones verdaderas** (obtenidas del predicado mediante la sustitución de argumentos del tipo apropiado por los indicadores de posición o parámetros del predicado). La **Suposición del Mundo Cerrado** dice que podemos dar por hecho que si una tupia válida —es decir, una que se apega al encabezado de la relación—no aparece en el cuerpo de la relación, entonces la proposición correspondiente *es falsa.*

Puesto que los valores de un tipo determinado pueden ser de cualquier clase, las relaciones que tienen **atributos con valor de relación** son válidas (y de hecho son muy útiles, como veremos en los capítulos 6 y 11).

Las **varrels** se presentan en dos variedades, varrels **base y vistas.** Vimos cómo definir varrels base (y cómo definir tipos para los atributos de dichas varrels) en **Tutorial D,** un lenguaje que usaremos para fines ilustrativos a lo largo de este libro.

Nota: Quizás haya notado que expusimos la cuestión de los operadores definidos por el usuario para los tipos escalares, pero no para los tipos de relación. La razón es que la mayoría de los operadores relacionales que necesitamos —restricción, proyección, junta, asignación relacional, etcétera— están de hecho integrados en el propio modelo y no requieren de una "definición por el usuario". (Lo que es más, dichas operaciones son *genéricas* en el sentido de que se aplican, en términos generales, a relaciones de todos los tipos.) Sin embargo, no hay razón por la que dichas operaciones integradas no se aumenten con un conjunto de operaciones definidas por el usuario cuando el sistema ofrezca un medio para definirlas.

Por último, esbozamos las características de SQL para definir "dominios" y tablas base (al estilo de SQL). En particular, señalamos que:

- Los "dominios" al estilo de SQL no son tipos.
- Las tablas al estilo de SQL (base u otras) no son relaciones, ya que —entre otras cosas— (a) permiten filas duplicadas y (b) tienen un ordenamiento de sus columnas de izquierda a derecha. De hecho, puede incluso haber dos o más columnas con el mismo nombre (aunque esta observación no se aplica a tablas SQL con nombre, es decir a tablas base o vistas). A manera de ejemplo, considere la tabla devuelta por la siguiente consulta de SQL:

```
SELECT * FROM
V, P ;
```

EJERCICIOS

5.1 Dado el diseño de la estructura del catálogo para la base de datos de departamentos y empleados que se muestra en la figura 3.6 del capítulo 3:

a. Cambie el nombre de los diversos componentes de ese catálogo de acuerdo con los términos relaciOnales formales presentados en este capítulo.

b. ¿Cómo podría ampliarse la estructura de ese catálogo para tomar en cuenta los tipos (dominios)?

c. Escriba una consulta con base en ese catálogo ampliado para obtener los nombres de todas las varrels que tengan un atributo de tipo EMP#.

d. ¿Cómo podría manejar la parte c de este ejercicio en un sistema SQL sin el soporte de tipos de datos definidos por el usuario?

5.2 ¿Sobre qué dominios están definidas las propias varrels del catálogo?

5.3 Dada la varrel base ESTRUCTURA_PARTES (para valores de ejemplo, consulte la figura 4.6 del capítulo 4):

a. Use las características de **Tutorial D** bosquejadas en este capítulo para escribir una definición de varrel y el correspondiente conjunto de definiciones de tipos.

b. Si suponemos que esta varrel está incluida en la base de datos de departamentos y empleados del ejercicio 5.1, muestre las actualizaciones que el sistema debe hacer al catálogo para reflejar su respuesta a la parte a.

c. Escriba un conjunto adecuado de operaciones DROP, en **Tutorial D,** para lograr que las actualizaciones al catálogo indicadas en la parte b, se deshagan.

5.4 Utilizando de nuevo las características de **Tutorial D** bosquejadas en este capítulo, escriba un conjunto adecuado de definiciones para la base de datos de proveedores, partes y proyectos de la figura 4.5 (vea los ejercicios del capítulo 4).

5.5 En la sección 5.2 señalamos que es estrictamente incorrecto decir que (por ejemplo) la cantidad de un cierto envío es 100 ("una cantidad es un valor de tipo CANT, no un valor de tipo INTEGER" Como consecuencia, la figura 4.5 (por ejemplo) es más bien imprecisa, en tanto que pareciera que es correcto pensar, por ejemplo, en las cantidades como si fueran enteros. Dada su respuesta al ejercicio
5.4, muestre la forma correcta de referirse a los diversos valores escalares de la figura 4.5.

5.6 Dada su respuesta al ejercicio 5.4, ¿cuáles de las siguientes expresiones escalares son
Para las válidas, indique el tipo del resultado; para las demás, muestre una expresión válida que logre lo que parece ser el efecto deseado.

a. `Y.CIUDAD = P.CIUDAD`

b. `PROYECTO | | PARTE`

C. `CANT * 100`

d. `CANT + 100`

e. `STATUS + 5`

f. `Y. CIUDAD < V. CIUDAD`

g. `COLOR • P.CIUDAD`

h. `Y.CIUDAD • P.CIUDAD II 'burg'`

5.7 Dé una definición de tipo admisible para un tipo escalar denominado CIRCULO. ¿Qué sel tores y operadores THE_ se aplican a este tipo? Además:

a. Defina un conjunto de operadores de sólo lectura para calcular el diámetro, la circunferencia y el área de un círculo dado.

b. Defina un operador de actualización para duplicar el radio de un círculo dado (para ser más pre cisos, para actualizar una variable CIRCULO dada de manera que no cambie su valor de circulo con excepción de que el radio sea el doble que antes).

5.8 En ocasiones, se sugiere que los dominios o tipos en realidad también son variables, como las varrels. Por ejemplo, los números de empleado válidos podrían crecer de tres a cuatro **dígitos a** forme se expande la empresa, de modo que podría ser necesario actualizar "el conjunto de todos los números de empleado posibles". Explique.

5.9 Una relación está definida para tener un *conjunto* de atributos y un *conjunto* de tupias. Alt bien, en matemáticas el conjunto vacío es perfectamente respetable; de hecho, generalmente es necesario que los resultados, teoremas, etcétera, que son ciertos para un conjunto de *n* elementos, **deban** seguir siéndolo si *n* = 0. ¿Puede una relación tener un conjunto vacío de tupias? ¿O un conjunto vacio
de atributos?

5.10 En ocasiones, se sugiere que una varrel en realidad es sólo un archivo tradicional de computadora, con "tupias" en vez de registros y "atributos" en vez de campos. Explique.

5.11 Como vimos, las operaciones de definición de datos ocasionan que se hagan actualizaciones al
catálogo. Pero el catálogo es sólo un grupo de varrels (al igual que el resto de la base de datos): por
lo tanto, ¿no podemos usar las operaciones normales de actualización INSERT, UPDATE y DELETE
para actualizar el catálogo en forma adecuada? Explique.

5.12 Escriba una secuencia de operaciones DROP, en **Tutorial D.** que tenga el efecto de destruí la información creada por el usuario en la base de datos de proveedores, partes y proyectos.

REFERENCIAS Y BIBLIOGRAFÍA

La mayoría de las siguientes referencias son aplicables a todos los aspectos del modelo relational, no sólo al aspecto estructural.

5.1 E. F. Codd: "A Relational Model of Data for Large Shared Data Banks", *CACM 13,* No. 6 (junio, 1970). Reeditado en *Milestones of Research—Selected Papers 1958-1982 (CACM 25th Anniversary Issue), CACM 26,* No. 1 (enero, 1983). Vea también la primera versión "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks", IBM Research Report RJ599 (agosto 19, 1969). *Nota:* Esa primera versión fue la primera publicación de Codd sobre el modelo relational.

El artículo que lo inició todo. Aunque con unos 30 años de antigüedad, asombrosamente se mantiene bien para su relectura. Desde luego, muchas de las ideas han sido redefinidas desde su primera publicación pero (por mucho) los cambios han sido de naturaleza evolutiva, no revolucionaria. De hecho, hay algunas ideas cuyas implicaciones todavía no se han explorado por completo.

Hacemos una observación en un pequeño aspecto de la terminología. En su artículo, Codd emplea el término "relaciones variables en el tiempo" en lugar del término que nosotros preferimos "variables de relación" (varrels). Aunque "relaciones variables en el tiempo" no es en realidad un muy buen término. Primero, las relaciones como tales, son *valores* y simplemente no "varían en el tiempo" (no hay ninguna notación matemática de una relación que tenga diferentes valores en momentos distintos). Segundo, si decimos por ejemplo en algún lenguaje de programación

```
DECLARE N INTEGER ;
```

no llamamos a N un "entero variable en el tiempo", la llamamos *variable entera.* Por lo tanto, en este libro mantendremos nuestra terminología "variable de relación", no la terminología "variable en el tiempo"; sin embargo, usted por lo menos debe estar al tanto de la existencia de esta última terminología.

5.2 E. F. Codd: *The Relational Model for Database Management Version 2.* Reading, Mass.: Addison-Wesley (1990).

Codd dedicó varios años a revisar y ampliar su modelo (al cual ahora se refiere como "el Modelo Relacional Versión 1" o RM/V1) y este libro es el resultado. Describe "el Modelo Relational Versión 2" (RM/V2). La diferencia esencial entre RM/V1 y RM/V2 es la siguiente: mientras que RM/V1 fue diseñado como un anteproyecto abstracto para un aspecto en particular del problema total de las bases de datos (en esencia el aspecto de la fundamentación), RM/V2 está diseñado como un plano abstracto para *todo el sistema.* De ahí que, mientras RM/V1 contenía sólo tres partes —estructura, integridad y manipulación— RM/V2 contenga 18; esas 18 partes, por supuesto, contienen no sólo a las tres originales, sino también partes que tienen que ver con el catálogo, la autorización, la asignación de nombres, las bases de datos distribuidas y muchos otros aspectos de la administración de bases de datos. Para fines de consulta, aquí tenemos una lista de las 18 partes:

A Autorización
B Operadores básicos
C Catálogo
D Principios de diseño de DBMS
E Comandos para el DBA
F Funciones
I Integridad
J Indicadores
L Principios de diseño de lenguaje
M Manipulación
N Asignación de nombres
P Protección
Q Calificadores
S Estructura
T Tipos de datos
V Vistas
X Base de datos distribuida
Z Operadores avanzados

Sin embargo, las ideas que promulga este libro no son de ningún modo aceptadas en fon universal [5.7-5.8]. Comentamos aquí un aspecto específico. Como vimos en el capítulo, los dominios limitan las comparaciones. Por ejemplo, en el caso de la base de datos de proveedores) partes, la comparación V.V# = VP.P# no es válida, debido a que los operandos son de tipos diferentes; por lo tanto, un intento de juntar números de proveedor y números de parte fracasará. Codd propone por lo tanto versiones de la **"anulación de comprobación de dominio" (I** de ciertas operaciones del álgebra relacional, la cual permite que las operaciones en cuestión se realicen aun cuando comprenden una comparación entre valores de diferentes tipos. Por ejemplo, una versión DCO de la junta recién mencionada, ocasionará que la junta se haga aun cuando los atributos V.V# y VP.P# sean de tipos diferentes (la junta se hace presuntamente sobre la base de *representaciones* coincidentes en lugar de *tipos* coincidentes).

Pero, por supuesto, es ahí donde reside el problema. *Toda la idea de DCO está basada en una confusión entre los tipos y las representaciones.* Reconocer los dominios por lo que son(decir, tipos) —con todo lo que implica este reconocimiento— nos da la comprobación de dominio que queremos, y también nos da algo como la posibilidad de DCO. Por ejemplo, la s guiente expresión constituye una comparación en el nivel de representación posible entre un número de proveedor y un número de parte:

```
THE_V#      V#       = THEP#     P#
```

(aquí, ambos operandos son de tipo CHAR). Por lo tanto, afirmamos que la clase de **mecanismo** expuesta en la sección 5.2 nos da todas las facilidades que queremos, pero lo hace de una manera clara, sistemática (es decir, no *ad hoc)* y completamente ortogonal. En particular, ahora no hay necesidad de saturar el modelo relacional con nuevos operadores como "junta DCO" (etcétera).

5.3 Hugh Darwen: "The Duplicity of Duplicate Rows", en C. J. Date y Hugh Darwen, i *Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Este artículo fue escrito para dar mayor soporte a los argumentos presentados en la referencia [5.6] en apoyo a la prohibición de las filas duplicadas. El artículo no sólo ofrece versiones fresca gunos de esos mismos argumentos, sino que también se las arregla para presentar otros adicionales. En particular, enfatiza la idea fundamental de que, con el fin de exponer de alguna manera inteli¡ la cuestión de si dos objetos están duplicados, es esencial tener un *criterio de igualdad* claro (de-nominado en el documento, criterio de *identidad)* para la clase de objetos bajo consideración. I palabras, ¿qué significa que dos objetos "sean lo mismo", ya sean filas en una tabla u otra cosa?

5.4 Hugh Darwen: "Relation-Valued Attributes", en C. J. Date y Hugh Darwen, *Relational Database*
Writings 1989-1991. Reading, Mass.: Addison-Wesley (1992).

5.5 Hugh Darwen: "The Nullologist in Relationland", en C. J. Date y Hugh Darwen, *Relatit base Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

La *nulología* es, como lo dice Darwen, "el estudio de nada en absoluto"; o en otras \ estudio del conjunto vacío. En la teoría relacional los conjuntos son ubicuos y la cuestión de qué sucede si un conjunto es vacío dista mucho de ser trivial. De hecho, muy a menudo r que el caso del conjunto vacío es absolutamente fundamental.

En lo que respecta al presente capítulo, las partes aplicables de este artículo son la sección. 2 ("Tablas sin filas") y la sección 3 ("Tablas sin columnas"). También vea la respuesta al cicio 5.9.

5.6 C. J. Date: "Why Duplicate Rows Are Prohibited", en *Relational Database Writings 1* Reading, Mass.: Addison-Wesley (1990).

Presenta una extensa serie de argumentos, con ejemplos, en apoyo a la prohibición de las filas duplicadas. En particular, el artículo muestra que las filas duplicadas constituyen un in *inhibidor de la optimización* (vea el capítulo 17). También vea la referencia [5.3].

5.7 C. J. Date: "Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1)", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Resume y critica el "RM/V1" de Codd [5.2] y ofrece una definición alternativa. La suposición es que resulta de crucial importancia asimilar la "Versión 1", antes de pensar siquiera en pasar a alguna "Versión 2". *Nota:* La versión del modelo relacional que se describe en el presente libro se basa en la versión "reconstituida" que se bosqueja en este artículo (y se clarifica y describe más a fondo en la referencia [3.3]).

5.8 C. J. Date: "A Critical Review of the Relational Model Version 2 (RM/V2)", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Resume y critica el "RM/V2" de Codd [5.2].

5.9 C. J. Date: "30 Years of Relational" (serie de doce artículos), *Intelligent Enterprise 1* números 1 a 3 e *Intelligent Enterprise 2* números 1 a 9 (octubre de 1998 en adelante). *Nota:* La mayoría de los fascículos, después del primero, están publicados en la versión en línea de la revista, en *www.intelligententerprise. com.*

Estos artículos se ofrecen como una revisión y valoración retrospectivas, cuidadosas y sin desviaciones, de la contribución relacional de Codd como está documentada en sus publicaciones de los años setenta. Para ser específicos, examinan en detalle los siguientes documentos (y de paso, también aborda otros más):

- Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks (la primera versión de la referencia [5.1]);
- A Relational Model of Data for Large Shared Data Banks [5.1];
- Relational Completeness of Data Base Sublanguages [6.1];
- A Data Base Sublanguage Founded on the Relational Calculus [7.1];
- Further Normalization of the Data Base Relational Model [10.4];
- Extending the Relational Database Model to Capture More Meaning [13.6];
- Interactive Support for Nonprogrammers: The Relational and Network Approaches [25.8].

5.10 Mark A. Roth, Henry F. Korth y Abraham Silberschatz: "Extended Algebra and Calculus for Nested Relational Databases", *ACM TODS 13,* No. 4 (diciembre, 1988).

A través de los años, varios investigadores han propuesto lo que se denominan "Relaciones NF^2", en donde NF^2 = NFNF = "no primera forma normal". En realidad existe cierta confusión sobre lo que significa exactamente el término NF^2, pero la interpretación que parece tener más sentido es ésta: Sea *nr* una "relación NF^2" y sea el "atributo" *A* de *nr* del tipo *T.* Entonces, una determinada "tupia" t de *nr* puede incluir cualquier cantidad de valores para el "atributo" *A* (¿incluso quizás ninguno?), y diferentes "tupias" pueden contener diferentes cantidades de dichos valores *A** (Por lo tanto, el atributo *A* es un atributo de **grupo de repetición;** vea las observaciones sobre este tema en la edición anterior de este libro, citada en la sección 5.3 del presente capítulo.) Por lo tanto, observe cuidadosamente, que la "relación" *nr* definitivamente *no es* normalizada (no contiene exactamente un valor para cada atributo en cada tupia); de hecho, por lo que respecta al modelo relacional, no es en absoluto una relación. Sea como fuere, este artículo expone una versión de la idea del NF^2. En particular, define un cálculo y un álgebra (vea los capítulos 6 y 7) para las "relaciones NF^2" y demuestra su equivalencia. Además, proporciona referencias a un cuerpo considerable de otro trabajo sobre el tema.

---

* Algunos autores rechazarían esta definición y definirían una "relación NF^2" para que fuera cualquier relación con al menos un atributo que tuviera el valor de una relación. Nosotros tenemos nuestras razones para argumentar que una relación así está, de hecho, en la primera forma normal.

RESPUESTAS A EJERCICIOS SELECCIONADOS

5.1

a. Los cambios obvios se resumen a continuación:

Reemplazar		*por*
TABLA		VARREL
		ATRIBUTO
NOMTABLA		NOMBREVR
CONTCOL		GRADO
CONTFTLA		CARDINALIDAD (a veces abreviado CARD)
NOMCOL		NOMATRIBUTO

Además, la varrel TABLA (ahora VARREL) necesita otro atributo (CLASEVR) cuyos valores indiquen si la varrel correspondiente es una varrel base o una vista. La estructura del catálogo luce ahora similar a ésta:

VARREL	NOMBREVR	GRADO	CARD	CLASEVR	
ATRIBUTO	NOMBREVR	NOMATRIBUTO	.....		

b. Necesitamos una nueva varrel del catálogo (TIPO) que contenga una entrada para cada tipo y un nuevo atributo (NOMTIPO) en la varrel ATRIBUTO que dé el tipo de cada atributo de todas las varrels. La estructura del catálogo ahora luce similar a la siguiente:

TIPO	NOMTIPO				
VARREL	NOMBREVR	GRADO	CARD	CLASEVR	
ATRIBUTO	NOMBREVR	NOMATRIBUTO	NOMTIPO	.....	

Como ejercicio adicional, tal vez deba considerar qué otras extensiones al catálogo serían necesarias para representar información con respecto a las claves primaria y externa.

c.
```
( ATRIBUTO WHERE NOMTIPO - NOMBRE ( 'EMP#' ) ) { NOMBREVR }
```

Aquí, observe la invocación al selector NOMBRE (vea la respuesta al ejercicio 5.2).

d. En un sistema que no soporta tipos definidos por el usuario, ¡obviamente no es posible const tar el catálogo con respecto a esos tipos!; sólo es posible consultarlo con respecto a los *atributos.* Sin embargo, hablando en términos generales, es una buena idea dar al atributo, cuando sea posible, el mismo nombre que el tipo correspondiente (incluso si el sistema no tiene cono-cimiento de dichos tipos) o por lo menos incluir ese nombre de tipo como (por decir algo) sufijo de su propio nombre. Si el definidor de la base de datos siguió esta convención de asignación de nombres, entonces un atributo de éstos quizás podría resolver el problema.

5.2 Obviamente, no es posible dar una respuesta definitiva a este ejercicio. Aquí hay algunas sugerencias razonables:

NOMTIPO	*está definido en el dominio*	NOMBRE
NOMBREVR		NOMBRE
NOMATRIBUTO		NOMBRE

GRADO	NONNEG_INTEGER
CARD	NONNEG_NTEGER
CLASEVR	CLASEVR

A su vez, damos por hecho que estos dominios se definen como sigue:

- NOMBRE es el conjunto de todos los nombres posibles.
- NONNEGJNTEGER es el conjunto de todos los enteros no negativos, menores a cierto límite superior definido por la implementación.
- CLASEVR es (en términos generales) el conjunto { "Base", "Vista" }.

¡Observe que en lo anterior violamos (parcialmente) nuestro propio principio!; es decir, el principio de que cada atributo debe tener, de ser posible, el mismo nombre que el tipo correspondiente. El ejercicio ilustra la idea de que estas violaciones tenderán a ocurrir si las varrels son diseñadas antes de haber fijado adecuadamente los tipos correspondientes (por supuesto, una observación que se aplica a todas las varrels, no sólo a las del catálogo).

5.3

a.
```
TYPE P#     POSSREP  ( CHAR ) ;
TYPE CANT POSSREP  (  INTEGER  ) ;

VAR ESTRUCTURA_PARTES BASE RELATION
       {  P#_MAYOR  P#,
          P#_MENOR P#,
          CANT      CANT } PRIMARY KEY {
        P#_MAYOR, P#_MENOR } ;
```

b. Sólo mostramos las entradas del catálogo (vea la figura 5.4). Observe que el valor CARD en la varrel VARREL de la propia VARREL, también necesitará incrementarse en uno. Además, mostramos la cardinalidad de ESTRUCTURA_PARTES como 0, no como 7 (a pesar del hecho de que la figura 4.6 muestra que contiene siete tupias). Esto se debe a que, presuntamente, la varrel estará vacía al crearla por primera vez.

c.
```
DROP VAR ESTRUCTURA_PARTES ;
DROP TYPE CANT ; DROP TYPE P# ;
```

TIPO

NOMTIPO	. .

VARREL

P# CANT	. . .

NOMBREVR	GRADO	CARD	CLASEVR	
ESTRUCTURA_PARTES	3	0	Base	

ATRIBUTO

NOMBREVR	NOMATRIBUTO	NOMTIPO	.....
ESTRUCTURA_PARTES ESTRUCTURA_PARTES ESTRUCTURA_PARTES	P#_MAY0R P#_MENOR CANT	P# P# CANT	.....

Figura 5.4 Entradas del catálogo para la varrel ESTRUCTURA_PARTES.

```
5.4 TYPE V# POSSREP ( CHAR )
    TYPE NOMBRE POSSREP ( CHAR )
                POSSREP ( CHAR ) ;
                POSSREP ( CHAR ) ;
                POSSREP ( RATIONAL )
                POSSREP ( CHAR ) ;
                POSSREP ( INTEGER
    TYPE P# TYPE COLOR TYPE PESO
    TYPE Y# TYPE CANT

    VAR V BASE RELATION { V#
         V#, PROVEEDOR
         NOMBRE, STATUS
         INTEGER, CIUDAD
         CHAR } PRIMARY KEY {
         V# } ;

    VAR P BASE RELATION
    { P#        P#,
      PARTE     NOMBRE,
      COLOR     COLOR,
      PESO      PESO,
      CIUDAD    CHAR }
    PRIMARY   KEY { P# }

VAR Y BASE RELATION

    { Y#         Y#,
      PROYECTO  NOMBRE,
      CIUDAD     CHAR }
    PRIMARY   KEY { Y# }

    VAR VPY BASE RELATION
          { V#      V#,
            P#      P#,
            Y#      Y#,
            CANT    CANT } PRIMARY KEY {
          V#, P#, Y# } FOREIGN KEY { V# }
          REFERENCES V FOREIGN KEY { P# }
          REFERENCES P FOREIGN KEY { Y# }
          REFERENCES Y
```

5.5 Mostramos un valor típico de cada atributo. Primero, la varrel V:

```
V#        :  V# ( 'V1 ' )
PROVEEDOR :  NOMBRE ( 'Smith'
STATUS    :  20
CIUDAD    : 'Londres'
```

Varrel P:

```
P#          :  P# ( 'P1 ' )
PARTE       :  NOMBRE (  'Tuerca' )
COLOR       :  COLOR (   Rojo' )
PESO        :  PESO (    .0 )
CIUDAD      : 'Londres'
```

Varrel Y:

```
Y#          : Y# (
              'Y1'
PROYECTO    :  NOMBRE (  'Clasificador'
CIUDAD      : 'París'
```

5.6

a. Válida; BOOLEAN.

b. No válida; NOMBRE (THEJVOMBRE(PROYECTO) II THE_NOMBRE (PARTE)). *Nota:* Aquí la idea es concatenar las (posibles) *representaciones de cadena de caracteres* y luego "con vertir" el resultado de esa concatenación de nuevo al tipo NOMBRE. Por supuesto, si el resul tado de la concatenación no puede convertirse a un nombre válido, la conversión misma cae en un error de tipo.

c. Válida; CANT.

d. No válida; CANT+ CANT(100).

e. Válida; STATUS.

f. Válida; BOOLEAN.

g. No válida; THE_COLOR(COLOR) = P.CIUDAD.

h. Válida.

```
5.7 TYPE LONGITUD POSSREP ( RATIONAL ) ;
    TYPE PUNTO   POSSREP ( X RATIONAL, Y RATIONAL ) ;
    TYPE CÍRCULO POSSREP ( R LONGITUD, CTRO PUNTO ) ¡
                /* R es (la longitud de) el radio del circulo */
                /* y CTRO es el centro                        */
```

El único selector que se aplica al tipo CIRCULO es como sigue:

```
CIRCULO ( r,  ctro )
/* regresa el círculo de radio r y centro ctro */
```

Los operadores THE_ son:

```
THE_R ( c )
/* regresa la longitud del radio del círculo c */

THE_CTRO ( c)
/* regresa el punto que es el centro del círculo c */
```

a.
```
OPERATOR DIAMETRO ( C CIRCULO ) RETURNS ( LONGITUD ) ;
   RETURN ( 2 * THE_R ( C ) ) ;
END OPERATOR ;

OPERATOR CIRCUNFERENCIA ( C CIRCULO ) RETURNS ( LONGITUD ) ;
   RETURN ( 3.14159 * DIAMETRO ( C ) ) ; END
OPERATOR ;

OPERATOR AREA ( C CIRCULO ) RETURNS ( AREA ) ;
RETURN ( 3.14159 * ( THE_R ( C ) ** 2 ) ) ; END
OPERATOR ;
```

Damos por hecho que (a) multiplicar una longitud por un entero o un racional regresa una longitud y (b) multiplicar una longitud por una longitud regresa un área (donde AREA es otro tipo definido por el usuario).

b.
```
OPERATOR DOBLE_R ( C CIRCULO ) UPDATES ( C ) ;
   BEGIN ;
      THE_R ( C )  := 2 * THE_R ( C ) ;
   RETURN ; END ;
END OPERATOR ;
```

5.8 Las siguientes observaciones son pertinentes. Primero, la operación de definir un tipo en realidad no crea el conjunto de valores correspondiente; de manera conceptual, dichos valores ya existen y siempre existirán (por ejemplo, piense en el tipo INTEGER). Por lo tanto, todo lo que en realidad hace la operación de "definir el tipo" —es decir, la instrucción TYPE, en **Tutorial** D— es introducir un *nombre* mediante el cual se puede hacer referencia a ese conjunto de valores. En forma similar, la instrucción DROP TYPE en realidad no elimina los valores correspondientes, simplemente elimina el nombre que introdujo la instrucción TYPE correspondiente. De aquí surge que esa "actualización de un tipo existente" en realidad significa eliminar el *nombre* del tipo existente y luego redefinir ese mismo no para hacer referencia a un conjunto de valores diferente. Por supuesto, no hay nada que impida el usi de algún tipo de abreviatura para "alterar el tipo" con el fin de simplificar esta operación.

5.9 Una relación con un conjunto vacío de tupias es perfectamente razonable y de hecho, es común (es análoga a un archivo sin registros). En particular, toda varrel base tiene como su valor inicial a una de estas relaciones, como señalé en la sección 5.4. Es común, aunque impreciso, referirse a este tipo de relación como una *relación vacía.*

Lo que tal vez sea menos obvio de manera inmediata, es que una relación con un conjunto de atributos vacío también es perfectamente razonable. De hecho, estas relaciones resultan ser de cru- *cial importancia;* en forma muy parecida a la importancia de los conjuntos vacíos en la teoría general de conjuntos o a la importancia del cero en la aritmética ordinaria.

Con el fin de examinar esta cuestión con un poco más de detalle, tenemos que considerar primero la cuestión de si una relación sin atributos puede contener alguna tupia. De hecho, dicha relación puede contener *como máximo una tupia,* es decir, la 0-tupla (la tupia sin componente alguno), y no puede contener más de una, ya que todas las 0-tuplas se duplican entre sí. Por lo tanto, existen precisamente dos relaciones de grado cero: una que contiene sólo una tupia y otra que no contiene tupia algunae absoluto. Estas dos relaciones son tan importantes que, siguiendo a Darwen [5.5], tenemos nombres especiales para ellas. A la primera le llamamos TABLE_DEE y a la segunda TABLE_DUM. o sola-mente DEE y DUM para abreviar (DEE es la que tiene una tupia, DUM es la vacía).

En este momento no es apropiado abundar en detalles sobre este tema; basta con decir que una razón por la que estas relaciones son tan importantes es que DEE corresponde a *verdadero* (o sí) DUM corresponde a *falso* (o *no).* Los ejercicios y respuestas de los capítulos 6 y 8 ofrecen un poco más de profundidad. Para una mayor explicación, consulte la referencia [5.5].

5.10 Podríamos acordar que una tupia sí se asemeja a un *registro* (ocurrencia, no tipo) y un **atributo** a un *campo* (tipo, no ocurrencia). Sin embargo, estas correspondencias son sólo aproximad; varrel no debe ser considerada como "sólo un archivo", sino más bien como un archivo diciplinado (vea la subsección "Propiedades de las relaciones" en la sección 5.3). La disciplina en cuestión que da como resultado una considerable simplificación de la estructura de los datos, tal como la ve el usuario, y por lo tanto una simplificación correspondiente en los operadores necesarios para tratar con los datos y en la interfaz de usuario en general.

5.11 En principio, la respuesta es sí, se *podría* actualizar el catálogo por medio de operaciones IN SERT, UPDATE y DELETE normales. Sin embargo, permitir dichas operaciones sería potencial- mente peligroso, ya que sería muy fácil destruir (en forma inadvertida o no) la informac catálogo que requiere el sistema para funcionar correctamente. Suponga, por ejemplo, que se tiera la operación DELETE

```
DELETE VARREL WHERE NOMBREVR - NOMBRE ( 'EMP' ) ;
```

en el catálogo de departamentos y empleados. Su efecto sería eliminar de la varrel VARREL a la tupla que describe a la varrel EMP. *En lo que respecta al sistema, la varrel EMP ya no existiría:* es

el sistema ya no tendría conocimiento de esa varrel. Por lo tanto, fracasarían todos los intentos subsiguientes por acceder a ella.

Por lo tanto, en la mayoría de los productos reales, las operaciones UPDATE, DELETE e INSERT del catálogo (a) no son permitidas en lo absoluto (el caso normal) o bien (b) son permitidas sólo para usuarios con autorización muy alta (quizás sólo para el DBA); en su lugar, las actualizaciones al catálogo se realizan por medio de instrucciones de *definición de datos.* Por ejemplo, definir la varrel EMP ocasiona (a) que se cree una entrada para EMP en la varrel VARREL y (b) que se cree un conjunto de cuatro entradas, una para cada atributo de EMP, en la varrel ATRIBUTO. (También hace que sucedan muchas otras cosas, sin embargo, por el momento no son de interés para nosotros.) Por lo tanto, definir un nuevo objeto —por ejemplo, un nuevo tipo o una nueva varrel base— es en cierta forma, el equivalente de INSERT para el catálogo. En forma similar, DROP es el equivalente de DELETE; y en SQL, el cual ofrece una variedad de instrucciones como ALTER TABLE (base), ALTER es el equivalente de UPDATE para modificar de distintas formas las entradas del catálogo.

Nota: Como vimos, el catálogo también incluye entradas para las propias varrels del catálogo. Sin embargo, esas entradas no se crean mediante operaciones explícitas de definición de datos. En su lugar, el propio sistema las crea automáticamente como parte de su proceso de instalación. En efecto, están "alambradas en forma permanente" dentro del sistema.

```
5.12 DROP VAR VPY, V, P, Y ;

     DROP TYPE V#, NOMBRE, P#, COLOR, PESO, Y#, CANT ;
```

Hemos tomado una forma extendida obvia de DROP VAR y DROP TYPE que permite eliminar varias variables o tipos en una sola operación.

CAPÍTULO 6

Álgebra relacional

6.1 INTRODUCCIÓN

La parte de manipulación del modelo relacional ha evolucionado considerablemente desde la publicación de los documentos originales de Codd sobre el tema [5.1,6.1]. Sin embargo, todavía se da el caso de que el componente principal de esa parte de manipulación es lo que se denomina **álgebra relacional,** que básicamente es sólo el conjunto de operadores que toman relaciones como sus operandos y regresan una relación como su resultado. En el capítulo 3 explicamos muy brevemente tres de estos operadores: *restringir, proyectar y juntar*. En el presente capítulo examinaremos dichos operadores a profundidad y otros más.

En la referencia [6.1], Codd definió lo que generalmente se conoce como álgebra "original" es decir, el conjunto de ocho operadores mostrados de manera simbólica en la figura 6.1. Ahora bien, Codd tenía en mente un propósito muy específico, el cual examinaremos en el siguiente capítulo para definir precisamente esos ocho operadores. Pero usted debe entender que éstos no son toda la historia; de hecho, podríamos definir cualquier cantidad de operadores que satisficieran el sencillo requerimiento de "relaciones dentro, relaciones fuera" y muchos de estos operadores han sido definidos por diferentes autores (por ejemplo, vea la referencia [6.10]). En este capítulo explicaremos primero los operadores originales de Codd (o al menos, nuestra versión de ellos) y los usaremos como base para explicar una variedad de ideas algebraicas; después consideraremos algunas formas en las que podría ampliarse ese conjunto original de ocho ope-radores.

Un panorama del álgebra original

El álgebra original [6.1] constaba de ocho operadores en dos grupos de cuatro cada uno:

1. El conjunto tradicional de operadores *unión, intersección, diferencia y producto cartesiano* (todos ellos modificados de alguna manera, para tomar en cuenta el hecho de que sus operandos son específicamente relaciones en lugar de conjuntos arbitrarios);

2. Los operadores relacionales especiales *restringir* (también conocido como *seleccionar proyectar, juntar* y *dividir*.

Aquí presentamos definiciones simplificadas de estos ocho operadores (consulte la figura6.1

Restringir: Regresa una relación que contiene todas las tupias de una relación especificada que satisfacen una condición especificada.

Proyectar: Regresa una relación que contiene todas las tupias o subtuplas que quedan en» relación especificada después de quitar los atributos especificados.

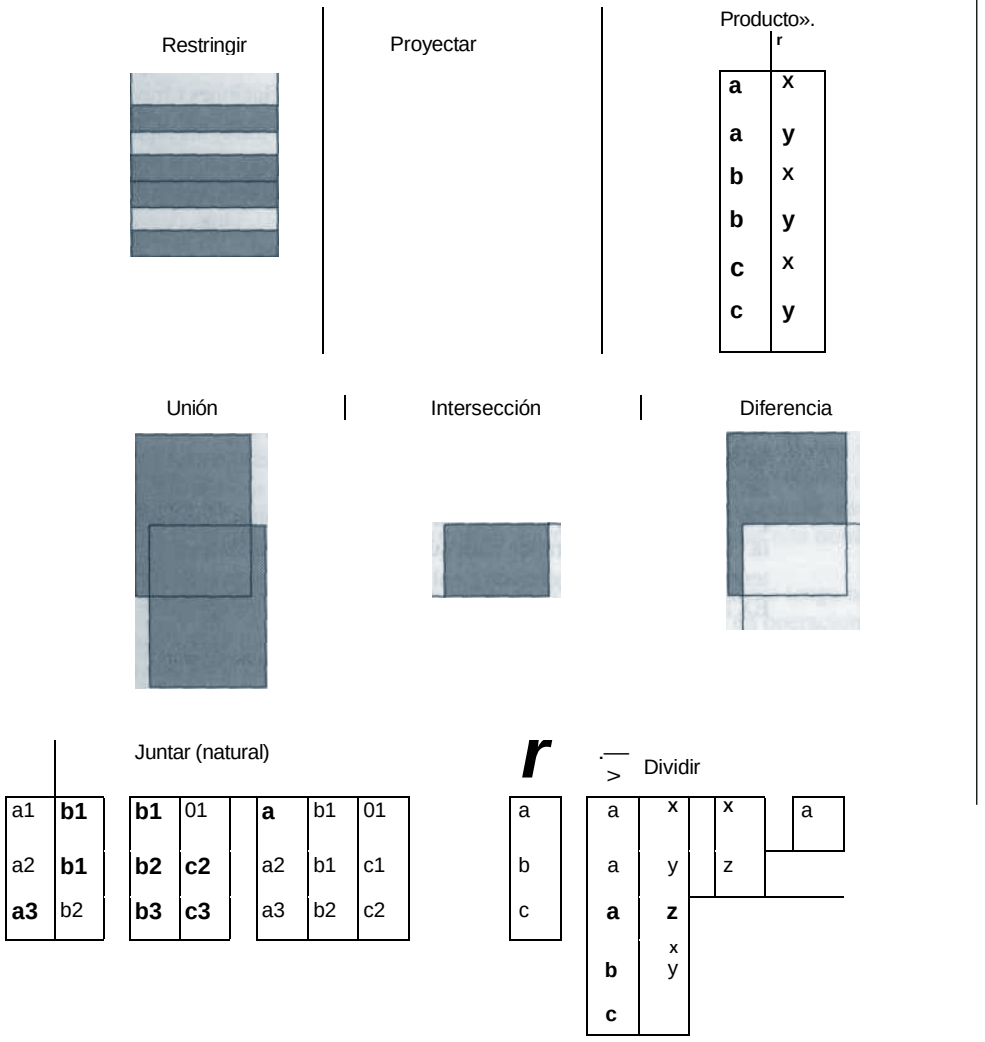


Figura 6.1 Panorama de los ocho operadores originales.

Producto: Regresa una relación que contiene todas las tupias posibles que son una combinación de dos tupias, una de cada una de dos relaciones especificadas.

Unión: Regresa una relación que contiene todas las tupias que aparecen en una o en las dos relaciones especificadas.

Intersección: Regresa una relación que contiene todas las tupias que aparecen en las dos **rela-**
ciones especificadas (en ambas, no en una u otra). **Diferencia:** Regresa
una relación que contiene todas las tupias que aparecen en la primera
pero no en la segunda de las dos relaciones especificadas.

Juntar: Regresa una relación que contiene todas las tupias posibles que son una coral nación de dos tupias de cada una de dos relaciones especificadas, tales que las dos tupias que contribuyen a cualquier combinación dada tengan un valor común para los atributos comunes de las dos relaciones (y ese valor común aparece sólo una vez, y no dos, en la tupia resultante).

Dividir: Toma dos relaciones unarías y una relación binaria y regresa una relación que contiene todas las tupias de una relación unaría que aparecen en la relación bi-naria y que a la vez coinciden con todas las tupias de la otra relación unaria.

Suficiente para nuestro breve repaso de los operadores originales. El plan para el resto del capítulo es el siguiente. Después de esta sección introductoria, la sección 6.2 explica una vez más la cuestión del cierre relacional y la explica detalladamente. Las secciones 6.3 y 6.4 explican en detalle los ocho operadores originales de Codd y la sección 6.5 ofrece algunos ejemplos de cómo pueden usarse esos operadores para formular consultas. A continuación, la sección 6.6 consider la cuestión más general de para qué sirve el álgebra. Después, la sección 6.7 describe varias ex-tensiones útiles al álgebra original de Codd, incluyendo en particular los importantes operadores EXTEND y SUMMARIZE. La sección 6.8 explica los operadores para hacer una correspondencia entre las relaciones que comprenden atributos con valor de relación y las que sólo compren-den atributos escalares. Posteriormente, la sección 6.9 considera las comparaciones relacionales. Por ultimo, la sección 6.10 ofrece un breve resumen.

Dos últimas observaciones preliminares:

- Por razones obvias, a menudo decimos cosas como *"X* es una restricción de *R"* (donde *R* específicamente una *varrel)* cuando lo que deberíamos decir en forma más correcta es *"*\ es una restricción de la relación que es el valor actual de la varrel *R"* o incluso "X es una va riable cuyo valor actual es una restricción de la relación que es el valor actual de la varrel R". Confiamos en que estas abreviaturas descuidadas no generarán confusión alguna.
- Dejamos para el capítulo 7 la explicación de los equivalentes de SQL para los operadora del álgebra relacional, por razones que se explican en ese capítulo.

6.2 REVISION DE LA PROPIEDAD DE CIERRE

Como vimos en el capítulo 3, al hecho de que la salida de cualquier operación relacional dada sea otra relación, se le conoce como la propiedad relacional de **cierre.** Para resumir, cierre **sig** nifica que podemos escribir **expresiones relacionales anidadas;** es decir, expresiones relacionales en las que los propios operandos están representados a su vez por expresiones relacionales de una complejidad potencialmente arbitraria. (Existe una obvia analogía entre la capacidad de anidar expresiones relacionales en el álgebra relacional y la capacidad de anid expresiones aritméticas en la aritmética ordinaria; además, el hecho de que las relaciones sean

cerradas bajo el álgebra es importante por las mismas razones por lo que lo es el hecho de que los números sean cerrados bajo la aritmética ordinaria.)

Ahora bien, cuando en el capítulo 3 explicamos el cierre, había una idea muy importante que evadimos deliberadamente. Recuerde que cada relación tiene dos partes: un **encabezado** y un **cuerpo.** En términos generales, el encabezado representa los atributos y el cuerpo las tupias. Ahora, el encabezado de una relación *base* —es decir, el valor de una varrel base— es obviamente bien entendido y conocido para el sistema, ya que está especificado como parte de la definición de la varrel base relacionada. Pero, ¿qué hay con las relaciones *derivadas?* Por ejemplo, considere la expresión

```
V JOIN P
```

(que representa la junta de proveedores y partes con base en ciudades coincidentes, en donde CIUDAD es el único atributo común a las dos relaciones). Sabemos cómo luce el cuerpo del resultado, pero ¿qué hay del encabezado? La propiedad de cierre dicta que el cuerpo debe *tener* un encabezado y el sistema necesita saber cuál es (desde luego también el usuario, como veremos en un momento). En otras palabras, es obvio que ese resultado debe ser de un **tipo de relación** bien definido. Por lo tanto, si vamos a tomar en serio la propiedad de cierre, es claro que debemos definir las operaciones relacionales de tal manera que garanticen que cada operación produzca un resultado con un tipo apropiado de relación, en particular con nombres de atributo apropiados.*

Por supuesto, una razón por la que requerimos que toda relación resultante tenga nombres de atributo apropiados es para poder *referirnos* a esos nombres de atributo en operaciones posteriores; en particular en operaciones que aparecen en cualquier parte dentro de la expresión anidada general. Por ejemplo, ni siquiera podríamos *escribir* una expresión como

```
( V JOIN P ) WHERE CIUDAD = 'Atenas'
```

si no supiéramos que el resultado de evaluar la expresión V JOIN P tenía un atributo llamado CIUDAD.

Por lo tanto, lo que necesitamos es un conjunto de **reglas de inferencia de tipo** (de relación) integradas dentro del álgebra, de tal modo que si conocemos los tipos de relación de las relaciones de entrada de cualquier operación relacional dada, podemos inferir el tipo (de relación) de salida de esa operación. Dadas dichas reglas, podemos inferir que cualquier *expresión* relacional (independientemente de lo compleja que sea) producirá un resultado que también tiene un tipo (de relación) bien definido y en particular, un conjunto bien definido de nombres de atributo.

Como un paso preliminar para lograr esta meta, presentamos un nuevo operador, RENAME, cuya finalidad es (en términos generales) renombrar los atributos dentro de una relación especificada. De manera más precisa, el operador RENAME toma una relación dada y regresa otra que es idéntica a la primera, con excepción de que por lo menos uno de los atributos tiene un nombre diferente. (Por supuesto, la relación dada se especifica por medio de una expresión

---

* Hacemos notar que este aspecto del álgebra ha sido pasado por alto en gran medida en la literatura (y también, lamentablemente, en el lenguaje SQL y por lo tanto en los productos SQL); con la notable excepción del tratamiento encontrado en Hall *et al.* [6.10] y en Darwen [6.2]. La versión del álgebra que se presenta en este capítulo tiene mucha influencia de estas dos referencias.

relacional que tal vez comprende otras operaciones relacionales). Por ejemplo, podriamos es-cribir:

```
V RENAME CIUDAD AS CIUDADV
```

Esta expresión —por favor, observe que *se trata* de una expresión y no de un "coma o instrucción, y por lo tanto puede estar anidada dentro de otras expresiones— produce una relación que tiene el mismo encabezado y cuerpo que la relación V, excepto que el atril ciudad se llama CIUDADV en lugar de CIUDAD:

V#	PROVEEDOR	STATUS	CIUDADV
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

Importante: Por favor observe que esta expresión RENAME *no* ha modificado la varrel de proveedores de la base de datos, es sólo una expresión, como lo es (por ejemplo) V JOIN VP y como lo es cualquier otra expresión que produce simplemente un cierto resultado (un re que en este caso en particular, resulta muy similar al valor actual de la varrel de proveedores).

Aquí tenemos otro ejemplo (esta vez, un "cambio de nombre múltiple"):

```
P RENAME PARTE AS NOMP, PESO AS PS
```

El resultado luce como el siguiente:

p#	NOMP	COLOR	PS	CIUDAD
P1	Tuerca	Rojo	12.0	Londres
P2	Perno	Verde	17.0	París
P3	Tornillo	Azul	17.0	Roma
P4	Tornillo	Rojo	14.0	Londres
P5	Leva	Azul	12.0	París
P6	Engrane	Rojo	19.0	Londres

Vale la pena señalar de manera explícita que la disponibilidad de RENAME significa que (a diferencia de SQL) el álgebra relacional no tiene necesidad de nombres de atributo ca dos como V.V#.

6.3 SINTAXIS

En esta sección presentamos una sintaxis concreta (básicamente la sintaxis de Tutorial D)p las expresiones del álgebra relacional. *Nota:* La mayoría de los textos de base de datos emplean una notación en cierto modo matemática o "griega" para los operadores relacionales, en donde por lo regular se utiliza 'o' para la restricción, *n* para la proyección, n para la intersección >< ("corbata de moño") para la junta, etcétera. Como ya habrá notado, preferimos usar pala

reservadas como JOIN y WHERE. Por supuesto, las palabras reservadas hacen expresiones más largas, pero creemos que también las hacen más fáciles para el usuario.

```
<expresión relacional>
    ::=   RELATION { <lista de expresiones de tupia separadas con comas de expresión
                      de tupla> }
          <nombre de varrel> <operación
          relacional> (  <expresión
          relacional>  )
```

Una <*expresión relacionate* es aquella que denota una relación (desde luego, indica un *valor* de relación). El primer formato es una *invocación al selector de relación* (como un caso especial incluye literales de relación); aquí no desglosamos en detalle la sintaxis de <*expresión de tupla*>, pero esperamos que los ejemplos sean suficientes para ilustrar la idea general. Se pretende que los otros formatos sean fáciles de entender.

```
<operación relacional>
    ::=   <proyectar> I <no proyectar>
```

En la sintaxis distinguimos <*proyectar*> de <*no proyectar*> por simples razones de prioridad de operadores (es conveniente asignar una prioridad alta a la proyección).

```
<proyectar>
    ::■ <expresión relacionala
                  { [ ALL BUT ] <lista de nombres de atributo separados con comas > }
```

La <*expresión relacional*> no debe ser <*no proyectar*>.

```
<no proyectar>
    ::=   <renombrar>
          <unión> I <intersección> I <diferencia> I <producto>
          <restringir> I <juntar> I <dividir>

<renombrar>
    ::=   <expresión  relacional>
                  RENAME <lista de cambios de nombre separados con comas>
```

La <*expresión relacional*> no debe ser <*no proyectar*>. Para la sintaxis de <*cambio de nombre*>, vea los ejemplos de la sección anterior.

```
<unión>
    ::= <expresión relacional>
                  UNION <expresión relacional>
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>, salvo que una o ambas puedan ser otra <*unión*>.

```
<intersección>
    ::= <expresión relacional>
                  INTERSECT <expresión relacional>
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>, salvo que una o ambas puedan ser otra <*intersección*>.

```
<diferencia>
    ::= <expresión relacional>
                  MINUS <expresión relacional>
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>.

```
<producto>
    ::=   <expresión relacional>
                  TIMES <expresión relacional>
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>, salvo que una o ambas puedan ser otro <*producto*>.

```
<restringir>
    ::= <expresión relacionala WHERE <expresión lógica>
```

La <*expresión relacional*> no debe ser <*no proyectar*>. *Nota:* La <*expresión lógica*> puede incluir una referencia a un atributo de la relación denotada por <*expresión relacional*> siempre que se permita una invocación a selector (por lo tanto, el ejemplo V WHERE CIUDAD = 'Londres' es un <*restringir*> válido).

```
<juntar>
    ::= <expresión relacional>
                  JOIN <expresión relacional>
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>, salvo que una o ambas puedan ser otro <*juntar*>.

```
<dividir>
    ::= <expresión relacional>
              DIVIDEBY <expresión relacional> PER <por>
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>.

```
<por>
    ::=    <expresión relacional>
        |  ( <expresión relacional>,  <expresión relacional> )
```

Las <*expresiones relacionales*> no deben ser <*no proyectar*>.

6.4 SEMÁNTICA

En esta sección explicamos la sintaxis de la sección 6.3 con algunos ejemplos. Consideran los operadores en la secuencia *unión, intersección, diferencia, producto, restringir, proyectar, juntar y dividir (renombrar* se cubrió en la sección 6.2).

Unión

En matemáticas, la unión de dos conjuntos es el conjunto de elementos que pertenecen ya sea a uno o a ambos conjuntos originales. Puesto que una relación es (o más bien, contiene) conjunto —para ser más específicos, un conjunto de tuplas—, obviamente es posible cons truir una unión de estos dos conjuntos; el resultado será un conjunto consistente en **toda:** tupias que aparecen en cualquiera o en ambas de las relaciones originales. Por ejemplo, unión del conjunto de tupias de proveedores que aparecen actualmente en la varrel \

conjunto de tupias de partes que aparecen actualmente en la varrel P, es en realidad un conjunto.

Sin embargo, aunque ese resultado es un conjunto, *no es una relación.* Las relaciones no pueden contener una mezcla de diferentes clases de tupias, deben ser "tupias homogéneas". Y por supuesto deseamos que el resultado sea una relación, ya que queremos conservar la propiedad de cierre. Por lo tanto, la unión en el álgebra relacional no es la unión matemática común; más bien es una clase especial de unión en la que requerimos que las dos relaciones de entrada sean **del mismo tipo;** esto significa que ambas deben contener, por ejemplo, tupias de proveedores o tupias de partes, pero no una mezcla de los dos. Si las dos relaciones son del mismo tipo (de relación), entonces podemos tomar su unión y el resultado será también una relación del mismo tipo (de relación); en otras palabras, se conservará la propiedad de cierre.*

Por lo tanto, aquí tenemos una definición del operador relacional de unión: Dadas dos relaciones *A* y *B* del mismo tipo, la **unión** de dichas relaciones (A UNION *B)* es una relación del mismo tipo con un cuerpo que consiste en todas las tupias í, tal que t aparece en A, en *B* o en ambas.

Ejemplo: Sean las relaciones *A* y *B* como muestra la figura 6.2 (ambas se derivan de la varrel de proveedores V; *A* son los proveedores en Londres y *B* son los proveedores que suministran la parte Pl, hablando de manera intuitiva). Entonces *A* UNION *B* —vea la parte a. de la figura— son los proveedores que están ubicados en Londres o bien, que suministran la parte Pl (o ambos). Observe que el resultado tiene tres tupias, no cuatro; por definición, las tupias duplicadas se eliminan. Observamos de paso que la única operación en la que también surge esta cuestión de los duplicados es la proyección (vea más adelante en esta sección).

Intersección

Al igual que la unión, y básicamente por la misma razón, el operador relacional de intersección requiere que sus operandos sean del mismo tipo. Dadas dos relaciones *A* y *B* del mismo tipo, entonces la **intersección** de esas dos relaciones *(A* INTERSECT *B)* es una relación del mismo tipo con un cuerpo que consiste en todas las tupias *t,* tal que *t* aparece tanto en *A* como en *B.*

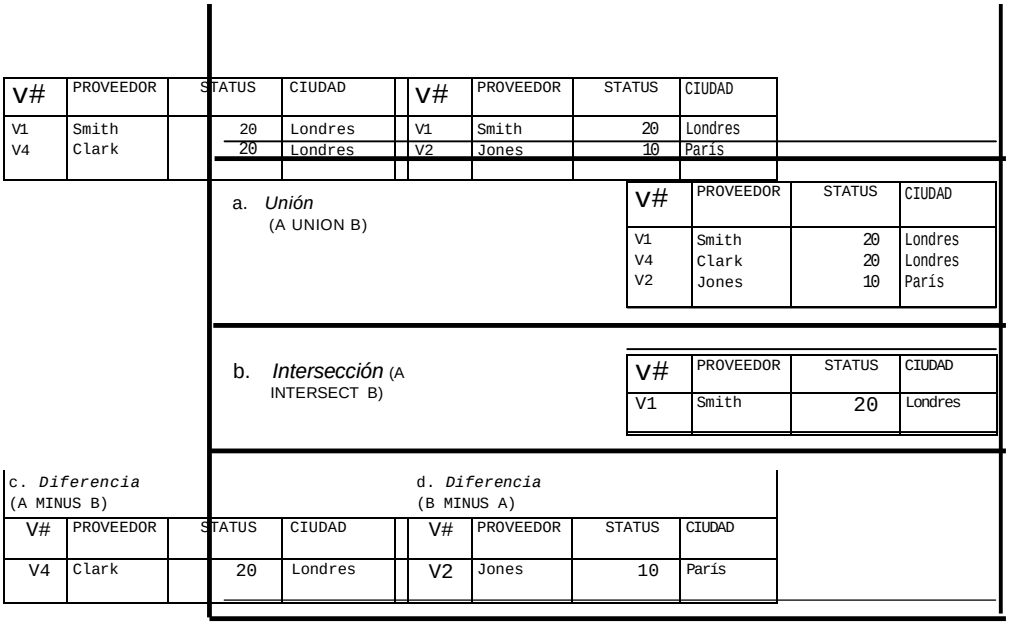
Ejemplo: Una vez más, sean *A* y *B* las relaciones que se muestran en la figura 6.2. Entonces *A* INTERSECT *B* —vea la parte b. de la figura— son los proveedores que están ubicados en Londres y que suministran la parte Pl.

Diferencia

Al igual que la unión y la intersección, el operador relacional de diferencia requiere también que sus operandos sean del mismo tipo. Dadas dos relaciones *A* y *B* del mismo tipo, entonces la **diferencia** entre esas dos relaciones *(A* MINUS *B,* en ese orden) es una relación del mismo tipo con un cuerpo que consiste en todas las tupias f, tal que *t* aparece en A y no en *B.*

---

* Históricamente, la mayor parte de la literatura sobre bases de datos (incluidas ediciones anteriores de este libro) usaban el término **compatibilidad con la unión** para referirse a la noción de que dos relaciones deben ser del mismo tipo. Sin embargo, este término no es muy adecuado por diversas razones, la más obvia de las cuales es que dicha noción no se aplica sólo a la unión.



V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V4	Clark	20	Londres

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París

a. *Unión* (A UNION B)

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V4	Clark	20	Londres
V2	Jones	10	París

b. *Intersección* (A INTERSECT B)

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres

c. *Diferencia* (A MINUS B)

V#	PROVEEDOR	STATUS	CIUDAD
V4	Clark	20	Londres

d. *Diferencia* (B MINUS A)

V#	PROVEEDOR	STATUS	CIUDAD
V2	Jones	10	París

Figura 6.2 Ejemplos de unión, intersección y diferencia.

Ejemplo: Sean de nuevo A y *B* las que muestra la figura 6.2. Entonces *A* MINUS *B* —vea la parte c. de la figura— son los proveedores que están ubicados en Londres y que no sumistran la parte Pl, mientras que *B* MINUS *A* —vea la parte d. de la figura— son los proveedor que suministran la parte Pl y que no están ubicados en Londres. Observe que MINUS tiene **una** direccionalidad, tal como la tiene la resta en la aritmética ordinaria (por ejemplo, "5 - 2" y "2 5" no son lo mismo).

Producto

En matemáticas, el producto cartesiano (por abreviar, producto) de dos conjuntos es el conju de todos los pares ordenados tales que en cada par, el primer elemento viene del primer conju to y el segundo elemento viene del segundo conjunto. Por lo tanto, el producto cartesiano de dos relaciones sería (a grandes rasgos) un conjunto ordenado de pares de *tupias.* Pero de nuevo queremos conservar la propiedad de cierre; en otras palabras, deseamos que el resultado contenga las tupias *como tales* y no *pares ordenados* de tupias. Por lo tanto, la versión relacional del pro-ducto cartesiano es una *forma ampliada* de la operación, en la cual cada par ordenado de tuplas es sustituido por una sola tupia que es la *unión* de las dos tupias en cuestión (usando "unió su sentido normal de la teoría de conjuntos, no en su sentido especial relacional). Esto es, dadas las tupias

A2:a2, Am:am

{ *B1:b1, B2:b2, ..., Bn:bn* } la

unión de ambas es la tupia sencilla

{ *A1:a1, A2:a2, ..., Am : am, B1:b* *1, B2:b2, ..., Bn:bn*

Otro problema que ocurre y está relacionado con el producto cartesiano es que (desde luego) requerimos que la relación resultante tenga un encabezado bien formado (es decir, que sea de un tipo apropiado de relación). Ahora bien, es claro que el encabezado del resultado consiste en todos los atributos de ambas relaciones de entrada. Por lo tanto, surgirá un problema si esos dos encabezados tienen un nombre de atributo común; si la operación fuera permitida, el encabezado del resultado tendría dos atributos con el mismo nombre y por lo tanto no estaría "bien formado". Entonces, si necesitamos generar el producto cartesiano de dos relaciones que tienen nombres de atributo comunes, primero debemos usar el operador RENAME para renombrar los atributos en forma adecuada.

Por lo tanto, definimos al **producto cartesiano** (relacional) de dos relaciones *A* y *B (A* TIMES *B)* —donde *A* y *B* no tienen nombres de atributo comunes— como una relación con un encabezado que es la unión (en la teoría de conjuntos) de los encabezados de *A* y *B* y con un cuerpo que consiste en el conjunto de todas las tupias *t,* tal que *t* es la unión (en la teoría de conjuntos) de una tupia que aparece en A y una tupia que aparece en *B.* Observe que la cardinalidad del resultado es el producto de las cardinalidades de *A* y *B, y* que la suma de sus grados es el grado del resultado.

Ejemplo: Sean las relaciones *A y B* como muestra la figura 6.3 *(A* son todos los números actuales de proveedores y *B* son todos los números actuales de partes, hablando de manera intuitiva). Entonces, *A* TIMES *B* (vea la parte inferior de la figura) son todos los pares número de proveedor/número de parte.

V#
V1
V2
V3
V4
V5

P#
P1
P2
P3
P4
P5
P6

Producto Cartesiano (A TIMES B)

V#	P#										
V1	P1	V2	P1	V3	P1	V4	P1	V5	P1
V1	P2	V2	P2	V3	P2	V4	P2	V5	P2
V1	P3	V2	P3	V3	P3	V4	P3	V5	P3
V1	P4	V2	P4	V3	P4	V4	P4	V5	P4
V1	P5	V2	P5	V3	P5	V4	P5	V5	P5
V1	P6	V2	P6	V3	P6	V4	P6	V5	P6

Figura 6.3 Ejemplo de un producto cartesiano.

Restringir

Tenga la relación *A* atributos *X y Y* (y quizás otros) y sea Ç un operador (por lo regular "=". ">", etcétera) tal que la **condición** *X © Y* esté bien definida y dé como resultado un valor de verdal *(verdadero o falso)* para valores particulares de *X y Y;* entonces la **restricción** 6 de la relación A sobre los atributos *Xy Y* (en ese orden) es una relación con el mismo encabezado que *A* y con un cuerpo que consiste en todas las tupias *t* de *A* tal que la condición *X* 0 *Y* dé como resultado *verdadero* para esa tupia *t.* Puntos a destacar:

1. Una invocación al selector (en particular, una literal) puede aparecer en la condición en lugar de *X* o de *Y* (o de ambas); de hecho, ese es el caso normal. Sin embargo, debemos ex-
   plicar que este "caso normal" es en realidad sólo una forma abreviada. Por ejemplo, la restricción

   ```
   V WHERE CIUDAD = 'Londres'
   ```

   es en realidad una forma abreviada de una expresión con la forma

   ```
   ( EXTEND V ADD 'Londres' AS TEMP ) WHERE CIUDAD = TEMP
   ```

   (donde el nombre TEMP es arbitrario). Vea la explicación de EXTEND en la sección

   6.7.

2. También es válida una condición de la forma *b* (donde *b* es una invocación al select lógico).

3. La restricción, tal como la hemos definido, permite una única condición en la cláusula WHERE. Sin embargo, y en virtud de la propiedad de cierre, es posible ampliarla que haya ambigüedad) a una forma en que la expresión de la cláusula WHERE consi en una combinación lógica cualquiera de dichas condiciones, gracias a las siguientes
   equivalencias:

   ```
   A WHERE C1 AND C2 = ( A WHERE C1 ) INTERSECT ( A WHERE C2 )

   A WHERE C1 OR C2 = ( A WHERE c1 ) UNION ( A WHERE C2 ) A

   WHERE NOT c    = A MINUS ( A WHERE C )
   ```

   Por lo tanto, de aquí en adelante daremos por hecho que la <*expresión lógica*> en la cláusula WHERE de una restricción consiste en una de estas combinaciones arbitrarias de diciones (de ser necesario, entre paréntesis para indicar el orden de evaluación deseado). donde a su vez, cada condición involucra atributos exclusivos de la relación pertinente o invocaciones al selector, o ambos. Observe que una <*expresión lógica*> como ésta, puede establecida como *verdadera o falsa* para una tupia dada, examinando sólo esa tupia de ma- nera aislada. Se dice que dicha <*expresión lógica*> es una **condición de restricción.**

El operador de restricción produce en efecto un subconjunto "horizontal" de una relación dada; es decir, ese subconjunto de tupias de la relación dada para el cual se satisface una condi-ción de restricción especificada. La figura 6.4 muestra algunos ejemplos.

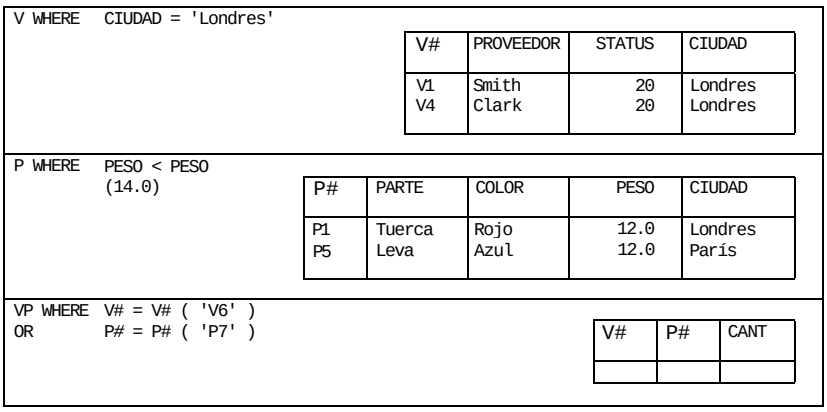


Figura 6.4 Ejemplos de restricción.

Proyectar

Tenga la relación *A* los atributos *X, Y,* ..., Z (y posiblemente otros); entonces la **proyección** de la relación *A* sobre *X, Y,* ..., Z

```
A { X, Y, . . . , Z }
```

es una relación con:

- Un encabezado derivado del encabezado de *A* al quitar todos los atributos no mencionados en el conjunto *{X,Y,...,Z},y*
- Un cuerpo consistente en todas las tupias *{X:x, Y:y,. ..,Z:z}* tal que una tupia aparece en *A* con el valor *x* para *X,* el valor *y* para *Y,* ..., y el valor z para *Z.*

Por lo tanto, el operador de proyección produce en efecto un subconjunto "vertical" de una relación dada; es decir, ese subconjunto obtenido al quitar todos los atributos no mencionados en la lista de atributos especificada y después al eliminar las (sub)tuplas duplicadas. Puntos a destacar:

1. Ningún atributo puede ser mencionado más de una vez en la lista con comas de nombre de atributo (¿por qué no?).
2. Si la lista de nombres de atributo separados con comas menciona todos los atributos de *A,* la proyección es una proyección *identidad.*
3. Es válida una proyección de la forma *A {}* (es decir, una proyección en la que la lista de nombres de atributo está vacía). Ésta representa una proyección de carácter *nulo.* Vea los ejercicios 6.8 al 6.10, al final del capítulo.

La figura 6.5 ofrece algunos ejemplos de proyección. Observe en el primer ejemplo (la proyección de proveedores sobre CIUDAD) que aunque la varrel V contiene actualmente cinco tupias, y por lo tanto cinco ciudades, sólo hay tres ciudades en el resultado. Los duplicados son

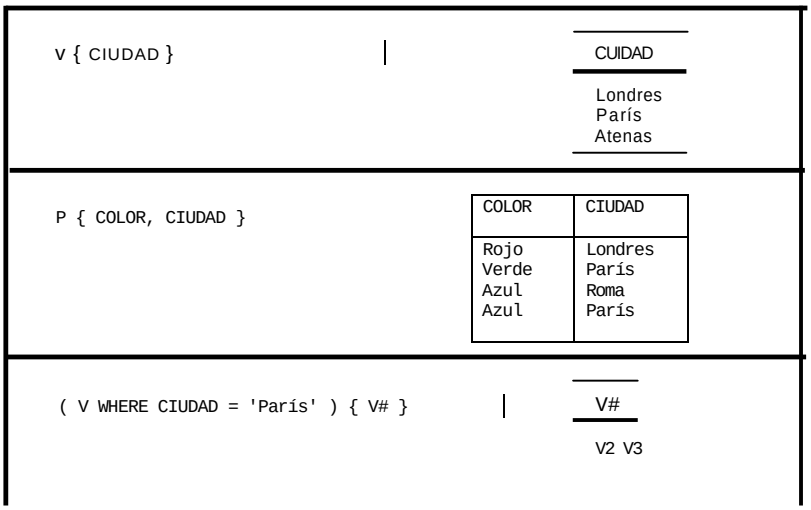


Figura 6.5 Ejemplos de proyección.

eliminados (esto es, las *tupias* duplicadas). Por supuesto, se aplican observaciones similars los otros ejemplos.

En la práctica, a menudo es conveniente poder especificar no los atributos sobre los que s va a tomar la proyección, sino más bien aquellos que se van a "proyectar hacia afuera" (es decir, eliminar). En lugar de decir, por ejemplo, "proyectar la relación P sobre los atributos P#, PARTE, COLOR y CIUDAD", podríamos decir "proyectar el atributo PESO hacia afuera de la relación P", como aquí:

```
P { ALL BUT PESO }
```

Juntar

Las juntas se presentan en diferentes variedades. Sin embargo, podemos decir que la más i portante es la llamada junta *natural;* tanto que de hecho, el término *junta* sin calificar se toma invariablemente para indicar específicamente la junta natural, y nosotros adoptamos ese uso en este libro. Entonces, aquí está la definición (es un tanto abstracta pero usted ya debe estar fami liarizado con la junta natural a un nivel intuitivo, por nuestras explicaciones en el capítulo 3 Tengan las relaciones *A* y *B* los encabezados

{ *X1, X2, Xm, Y1, Y2, Yn* }

{ *Y1, Y2, ..., Yn, 21, Z2, . . . , Zp* }

respectivamente; es decir, (sólo) los atributos *Y: Yl, Y2, ..., Yn* son comunes a las dos rela-
ciones. Los atributos *X: XI, X2, ..., Xm* son los otros atributos de A y los atributos Z: *Zl, Z2* .....

Zp son los otros atributos de *B.* Ahora, considere que *{Xl,X2,...,Xm}, {Yl,Y2,...,Yn}* y *{Zl,Z2,...,Zp}* son los tres atributos compuestos *X, Y y Z,* respectivamente. Entonces, la **junta natural** de *A y B*

```
A JOIN S
```

es una relación con el encabezado *{X,Y,Z}* y un cuerpo que consiste en el conjunto de todas las tupias *{X:x,Y:y,Z:z},* tal que una tupia aparece en *A* con el valor *x* para X y el valor *y* para *Y,* mientras que una tupia aparece en *B* con el valor *y* para *Y y* el valor z para Z.

La figura 6.6 presenta un ejemplo de una junta natural (la junta natural V JOIN P, sobre el atributo común CIUDAD).

V#	PROVEEDOR	STATUS	CIUDAD	P#	PARTE	COLOR	PESO
V1	Smith	20	Londres	P1	Tuerca	Rojo	12.0
V1	Smith	20	Londres	P4	Tornillo	Rojo	**14.0**
V1	Smith	20	Londres	P6	Engrane	Rojo	**19.0**
V2	Jones	10	Paris	P2	Perno	Verde	17.0
V2	Jones	10	Paris	P5	Leva	Azul	12.0
V3	Blake	30	París	P2	Perno	Verde	**17.0**
V3	Blake	30	París	P5	Leva	Azul	**12.0**
V4	Clark	20	Londres	P1	Tuerca	Rojo	12.0
V4	Clark	20	Londres	P4	Tornillo	Rojo	**14.0**
V4	Clark	20	Londres	P6	Engrane	Rojo	19.0

Figura 6.6 La junta natural V JOIN P.

Nota: Hemos ilustrado la idea varias veces —de hecho, está ilustrada por la figura 6.6— pero todavía cabe mencionar de manera explícita que las juntas *no* siempre son entre una clave externa y una clave primaria coincidente, aunque dichas juntas son muy comunes y un caso especial importante.

Pasemos ahora a la operación junta ç. Esta operación está diseñada para aquellas ocasiones (en comparación raras, pero de ningún modo desconocidas) en donde necesitamos juntar dos relaciones con base en algún operador de comparación distinto al de la igualdad. Sean las relaciones *A y B* tales que satisfagan los requerimientos del producto cartesiano (es decir, no tengan nombres de atributo comunes); tenga *A* un atributo *X y B* un atributo *Y,* y sean *X, Y, y* 0 tales que satisfagan los requerimientos de la restricción. Entonces la **junta** 0 de la relación *A* sobre el atributo *X* con la relación *B* sobre el atributo *Y,* se define como el resultado de evaluar la expresión

```
( A TIMES S ) WHERE X 0 Y
```

En otras palabras, es una relación con el mismo encabezado que el producto cartesiano de *A y B,* y con un cuerpo que consiste en el conjunto de todas las tupias *t,* tal que *t* aparece en ese producto cartesiano y la condición *"X* 0 *Y"* resulta *verdadera* para esa tupia *t.*

A manera de ejemplo, suponga que deseamos calcular la *junta mayor-que* de la relación V sobre CIUDAD con la relación P sobre CIUDAD (puesto que aquí 0 es ">", asumimos que ">" tiene sentido para las ciudades y lo interpretamos como que simplemente significa "mayor en orden alfabético"). Una expresión relacional apropiada es la siguiente:

```
( ( V RENAME CIUDAD AS CIUDADV ) TIMES (
  P RENAME CIUDAD AS CIUDADP ) )
                    WHERE CIUDADV > CIUDADP
```

Observe el cambio de nombre de atributos en este ejemplo. (Por supuesto, hubiera sido ficiente renombrar sólo uno de los dos atributos CIUDAD; la única razón para renombrar ara es la simetría.) La figura 6.7 muestra el resultado de la expresión general.

Si © es "=", la junta se denomina **equijunta.** Se sigue de la definición que el resultad de una equijunta debe incluir dos atributos con la propiedad de que los valores de esos dos al butos son iguales en cada tupia de la relación. Si uno de esos dos atributos se proyecta hac afuera y el otro se renombra (en caso necesario) de manera adecuada, ¡el resultado es la ju natural! Por ejemplo, la expresión que representa la junta natural de proveedores y partes (s las ciudades)

```
V JOIN P
```

es equivalente a la siguiente expresión más compleja:

```
( ( V TIMES ( P RENAME CIUDAD AS CIUDADP ) )
                    WHERE CIUDAD = CIUDADP )
                                { ALL BUT CIUDADP }
```

Nota: **Tutorial D** no incluye soporte directo para el operador junta 0 ya que (a) en la práctica no se necesita con mucha frecuencia y en todo caso, (b) no es un operador primitivo, cot hemos visto.

V#	PROVEEDOR	STATUS	CIUDADV	P#	PARTE	COLOR	PESO	CIUDADP
V2	Jones	10	París	P1	Tuerca	Rojo	12.0	Londres
V2	Jones	10	París	P4	Tornillo	Rojo	14.8	Londres
V2	Jones	10	París	P6	Engrane	Rojo	19.8	Londres
V3	Blake	30	París	P1	Tuerca	Rojo	12.8	Londres
V3	Blake	30	París	P4	Tornillo	Rojo	14.8	Londres
V3	Blake	30	París	P6	Engrane	Rojo	19.0	Londres

Figura 6.7 La junta "mayor que" de proveedores y partes sobre ciudades.

Dividir

La referencia [6.3] define dos operadores distintos para "dividir" que denomina División Pe queña y División Grande, respectivamente. En **Tutorial D,** una <*división*> en la que el <por> consiste en una sola <*expresión relacionate* es una División Pequeña; una <*división*> en la que <*por*> consiste en una lista separada con comas (entre paréntesis) de dos <*expresiones relacionales*> es una División Grande. La descripción que sigue se aplica sólo a la División Pequeña y sólo a una forma limitada en particular dentro de ésta. Vea la referencia [6.3] para ver una ex-plicación de la División Grande, así como para obtener detalles adicionales con respecto ala visión Pequeña.

Debemos decir también que la versión de la División Pequeña tal como la explico aquí no es la misma que el operador original de Codd; de hecho, es una versión mejorada que supera

ciertas dificultades que surgieron con ese operador original para las relaciones vacías. Tampoco es la misma que expliqué en las primeras ediciones de este libro.

He aquí la definición. Tengan las relaciones *A* y *B* los encabezados

```
X2,        Xm }
```

```
{ Y1, Y2, ..., Yn }
```

respectivamente (es decir, *A y B* tienen encabezados disjuntos); y tenga la relación *C* el encabezado

```
{ X1, X2, ..., Xm, Y1, Y2, ..., Yn }
```

(es decir, *C* tiene un encabezado que es la unión de los encabezados *de A* y *B).* Ahora, consideremos a *{XI,X2,... ,Xm} y a {Yl,Y2,...,Yn)* como los atributos *compuestos X* y *Y,* respectivamente. Entonces, la **división** de *A* entre *B* por *C* (donde *A* es el dividendo, *B* es el divisor y *C* es el "mediador")

```
A DIVIDEBY 8 PER C
```

es una relación con un encabezado *{X} y* un cuerpo que consiste en todas las tupias *{X:x},* tal que una tupia *{X:x,Y:y}* aparece en C para todas las tupias *{Y:y}* que aparecen en *B.* En otras palabras, el resultado consiste en aquellos valores *X* de *A* cuyos correspondientes valores *Y* en *C* incluyen a todos los valores *Y* de *B,* en términos generales.

La figura 6.8 muestra ejemplos sencillos de división. En cada caso, el dividendo (DEND) es la proyección del valor actual de la varrel V sobre V#; el mediador (MED) es en cada caso la proyección del valor actual de la varrel VP sobre V# y P#; los tres divisores (DOR) son como indica la figura. Observe en particular el último ejemplo, en el que el divisor es una relación que contiene los números de parte de todas las partes conocidas actualmente; el resultado muestra (obviamente) los números de proveedores que suministran esas partes. Como sugiere este ejemplo, el operador DIVIDEBY está diseñado para consultas de esta misma naturaleza general; de hecho, siempre que la versión en lenguaje natural de la consulta contiene la palabra "todos" ("obtener los proveedores que suministran *todas* las partes"), existe una gran posibilidad de que la división esté involucrada.* Sin embargo, vale la pena señalar que a menudo dichas consultas se expresan con mayor claridad en términos de comparaciones relacionales. (Vea la sección 6.9.)

Asociatividad y conmutatividad

Es fácil verificar que UNION es **asociativa;** es decir, si *A, B y C* son expresiones relacionales cualesquiera que producen relaciones del mismo tipo, entonces las expresiones

```
( A UNION B ) UNION C
```

*De hecho, Codd diseñó específicamente la división para ser una contraparte algebraica del *cuantificador universal*—vea el capítulo 7— en forma muy parecida a como fue diseñada la proyección para ser una contraparte algebraica del cuantificador *existential.*

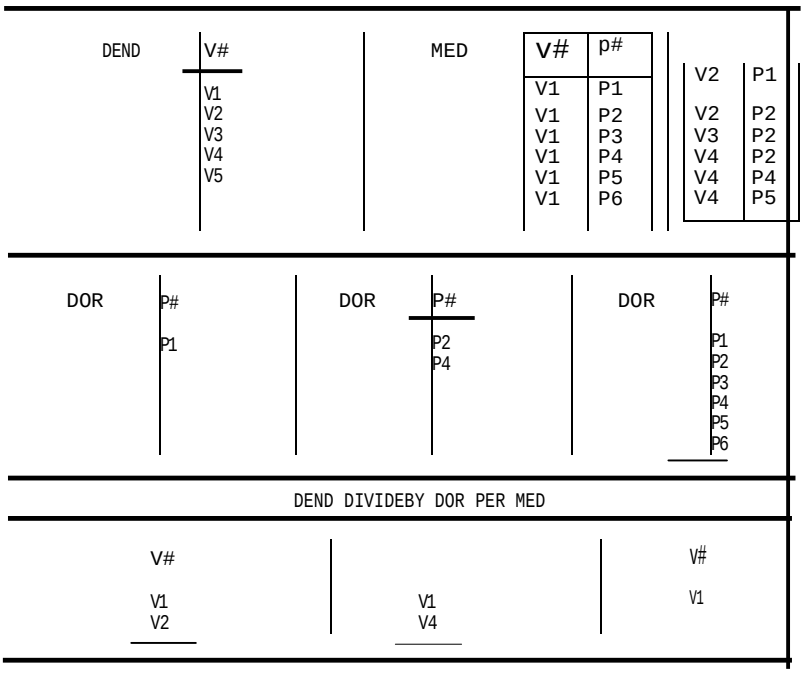


Figura 6.8 Ejemplos de división.

A UNION (S UNION *C*

son lógicamente equivalentes. Por lo tanto, por comodidad permitimos que se escriba una se-cuencia de UNIONs sin paréntesis alguno; es decir, cada una de las expresiones ante podría simplificarse sin ambigüedad a solamente

A UNION S UNION C

Se aplican observaciones similares para INTERSECT, TIMES y JOIN (aunque no MINUS).

Señalamos además que UNION, INTERSECT, TIMES y JOIN (aunque no MINUS; bién son **conmutativas;** es decir, las expresiones

A UNION S

B UNION *A*

también son equivalentes lógicamente, y en forma similar también INTERSECT, TIMES y JOIN. *Nota:* En el capítulo 17 revisaremos por completo la cuestión de la asociatividad y la conmutatividad. Por cierto, con respecto a TIMES, cabe resaltar que la operación de producto cartesiano de la teoría de conjuntos no es asociativa ni conmutativa, pero (como acabamos de ver) la versión relacional sí presenta ambas propiedades.

Por último, observamos que si *A* y *B* no tienen nombres de atributo comunes, entonces *A* JOIN B es equivalente a A TIMES *B* [5.5]; es decir, en este caso la junta natural degenera en un producto cartesiano. De hecho, por esta precisa razón la versión de **Tutorial D** definida en la referencia [3.3] no incluye un manejo directo del operador TIMES.

6.5 EJEMPLOS

En esta sección presentamos algunos ejemplos sobre el uso de expresiones del álgebra relacional en la formulación de consultas. Le recomendamos que compruebe estos ejemplos con los datos de muestra de la figura 3.8.

6.5.1 Obtener los nombres de los proveedores que suministran la parte P2.

```
( ( VP JOIN V ) WHERE P# = P# ( 'P2' ) ) { PROVEEDOR >
```

Explicación: Primero se genera la junta natural de VP y V sobre los números de proveedor, lo cual tiene el efecto —conceptualmente— de extender cada tupia de VP con la información del proveedor correspondiente (es decir, con los valores PROVEEDOR, STATUS Y CIUDAD correspondientes). Después, esa junta se restringe sólo a las tupias de la parte P2. Por último, esa restricción se proyecta sobre PROVEEDOR. El resultado final sólo tiene un atributo llamado PROVEEDOR.

6.5.2 Obtener los nombres de los proveedores que suministran por lo menos una parte roja.

```
( ( ( P WHERE COLOR = COLOR ( 'Rojo' ) )
                    JOIN VP ) { V# } JOIN V ) { PROVEEDOR }
```

Nuevamente, el único atributo del resultado es PROVEEDOR.

Por cierto, aquí tenemos una formulación diferente de la misma consulta:

```
( ( ( P WHERE COLOR = COLOR ( 'Rojo' ) ) { P# }
                    JOIN VP ) JOIN V ) { PROVEEDOR }
```

Por lo tanto, el ejemplo ilustra la importante idea de que a menudo existirán varias formas diferentes de formular una determinada consulta. Consulte el capítulo 17 para una explicación de algunas de las implicaciones de esta idea.

6.5.3 Obtener los nombres de los proveedores que suministran todas las partes.

```
( ( V { V# } DIVIDEBY P { P# } PER VP { V#, P# } )
                                          JOIN V ) { PROVEEDOR }
```

Una vez más, el resultado tiene un único atributo llamado PROVEEDOR.

6.5.4 Obtener los números de los proveedores que suministran al menos todas las partes **que suministra el proveedor V2.**

```
V { V# } DIVIDEBY ( VP WHERE V# = V# ( ' V2' ) ) { P# }
                                                PER VP { V#, P# }
```

El resultado tiene un solo atributo denominado V#.

6.5.5 Obtener todos los pares de números de proveedor tales que los dos proveedores en **cuestión estén "coubicados" (es decir, localizados en la misma ciudad).**

```
( ( ( V RENAME V# AS VA ) { VA, CIUDAD } JOIN (
    V RENAME V# AS VB ) { VB, CIUDAD } ) WHERE
    VA < VB ) { VA, VB }
```

El resultado tiene dos atributos llamados VA y VB (por supuesto, hubiera sido suficiente con renombrar sólo uno de los dos atributos V#; lo hicimos con ambos por simetría). *Nota:* Hemos dado por hecho que el operador "<" ha sido definido para el tipo V#. La finalidad de la condición VA < VB es doble:

Elimina los pares de números de proveedor de la forma *(x,x);* Garantiza que no aparezcan ambos pares *(x,y)* y *(y,x).*

Mostramos otra formulación de esta consulta con el fin de ilustrar el uso de WITH para pre-sentar nombres abreviados de expresiones y simplificar así la tarea de escribir consultas extensas. (De hecho, anteriormente ilustramos WITH en el capítulo 5, sección 5.2 en la subsección "Definición de operadores").

```
WITH ( V RENAME V# AS VA ) { VA, CIUDAD } AS T1,
     ( V RENAME V# AS VB ) { VB, CIUDAD } AS T2,
     T1 JOIN T2 AS T3, T3 WHERE VA < VB AS T4 :
     T4 { VA, VB }
```

WITH nos permite pensar en expresiones extensas y complicadas paso a paso, y además no viola de ninguna manera la característica de no procedimientos del álgebra relaciona). Ahon-daremos en esta idea en la explicación que presentamos después del siguiente ejemplo.

6.5.6 Obtener los nombres de los proveedores que no suministran la parte P2.

```
( ( V { V# } MINUS ( VP WHERE P# = P# ( ' P2' ) ) { V# } )
                                              JOIN V ) { PROVEEDOR }
```

El resultado tiene un único atributo denominado PROVEEDOR.

Como prometimos, ahondaremos en este ejemplo con el fin de ilustrar otra idea. No siempre es fácil ver de inmediato cómo formular una consulta dada como una sola expresión dada. Ni tampoco debe ser necesario hacerlo. Aquí tenemos la formulación paso a paso del ejemplo 6.5.6:

```
WITH V { V# } AS T1,
     VP WHERE P# = P# ( 'P2' ) AS T2,
     T2 { V# } AS T3,
     T1 MINUS T3 AS T4,
     T4 JOIN V AS T5,
     T5 { PROVEEDOR } AS T6 :
     T6
```

T6 denota el resultado deseado. *Explicación:* Los nombres introducidos por una cláusula WITH —es decir, los nombres de la forma 17, en el ejemplo— se toman como locales para la instrucción que contiene dicha cláusula. Ahora bien, si el sistema maneja la "evaluación diferida" (como lo hizo, por ejemplo, el sistema PRTV [6.9]) entonces dividir de esta forma la consulta general en una secuencia de pasos necesita que *no* existan implicaciones indeseables de rendimiento. En su lugar, la consulta puede ser procesada como sigue:

- Las expresiones que anteceden a los dos puntos no requieren de una evaluación inmediata por parte del sistema, todo lo que éste tiene que hacer es recordarlas junto con los nombres introducidos por las cláusulas AS correspondientes.
- La expresión que sigue a los dos puntos denota el resultado final de la consulta (en el ejem plo, esa expresión es simplemente "T6"). Al llegar a este punto, el sistema no puede de morar más la evaluación sino que en vez de ello debe calcular de alguna forma el valor deseado (es decir, el valor de T6).
- Con el fin de evaluar T6, que es la proyección de T5 sobre PROVEEDOR, el sistema debe primero evaluar T5; para poder evaluar T5, que es la junta de T4 y V, el sistema debe antes evaluar T4; y así sucesivamente. En otras palabras, el sistema tiene efectivamente que eva luar la expresión anidada original, exactamente como si el usuario la hubiese escrito en primer lugar.

Vea en la siguiente sección una breve explicación de la cuestión general de evaluar dichas expresiones anidadas, así como el capítulo 17 para un tratamiento más amplio del mismo tema.

6.6 ¿PARA QUÉ SIRVE EL ÁLGEBRA?

Para resumir hasta este momento el capítulo, hemos definido un *álgebra relacional;* es decir, un conjunto de operaciones sobre relaciones. Las operaciones en cuestión son unión, intersección, diferencia, producto, restricción, proyección, junta y división, más un operador para renombrar atributos, RENAME (éste es en esencia el conjunto de operadores que definió Codd original-mente en la referencia [6.1], con excepción de RENAME). También hemos presentado una sintaxis para dichas operaciones y hemos utilizado esta sintaxis como base para varios ejemplos e ilustraciones.

Sin embargo, como quedó implícito en nuestra explicación, los ocho operadores de Codd no constituyen un conjunto *mínimo* (ni pretendieron serlo), ya que algunos de ellos no son primitivos; pueden ser definidos en términos de los otros. Por ejemplo, los tres operadores juntar, intersección y dividir pueden ser definidos en términos de los otros cinco (vea el ejercicio 6.2). Puesto que ninguno de esos otros cinco —restringir, proyectar, producto, unión y diferen-

cia—* pueden ser definidos en términos de los cuatro restantes (vea el ejercicio 6.4), podemos considerarlos como un conjunto **primitivo** o mínimo (por supuesto, no necesariamente el único. Sin embargo, en la práctica los otros tres operadores (en especial el de juntar) son tan útiles que es posible hacer un buen caso para manejarlos en forma directa.

Ahora estamos en posición de aclarar otra idea importante. Aunque nunca lo **dijimos de** manera explícita, hasta ahora el cuerpo del capítulo ha sugerido ciertamente que la finalidad principal del álgebra es simplemente la *recuperación de datos.* Sin embargo, ése no es el caso. intención fundamental del álgebra es permitir **la escritura de expresiones relacionales.** A su **vez,** dichas expresiones pretenden atender diversos fines, incluyendo desde luego la recuperación pero no están limitadas a ese solo propósito. La siguiente lista —que no pretende ser completa-indica algunas aplicaciones posibles de dichas expresiones:

- Definir un alcance para la **recuperación;** es decir, definir los datos a obtener en alguna operación de recuperación (como ya explicamos ampliamente);
- Definir un alcance para la **actualización;** es decir, definir los datos a insertar, modificar o eliminar en alguna operación de actualización (vea el capítulo 5);
- Definir **restricciones de integridad;** es decir, definir ciertas restricciones que la base de datos debe satisfacer (vea el capítulo 8);
- Definir **varrels derivadas;** es decir, definir los datos a incluir en una vista o instant (vea el capítulo 9);
- Definir **requerimientos de estabilidad;** es decir, definir los datos que serán el alcana cierta operación de control de concurrencia (vea el capítulo 15);
- Definir **restricciones de seguridad;** es decir, definir los datos sobre los cuales se va a conceder autorización de alguna clase (vea el capítulo 16).

De hecho, las expresiones sirven generalmente como *una representación simbólica de alto nivel de la intención del usuario* (por ejemplo, con respecto a alguna consulta en particular). precisamente debido a que son simbólicas y de alto nivel, pueden ser manipuladas de acue con una variedad de **reglas de transformación** simbólicas de alto nivel. Por ejemplo, la ex-presión

```
( ( VP JOIN V ) WHERE P# = P#      'P2'       ) { PROVEEDOR }
```

("nombres de los proveedores que suministran la parte P2"; ejemplo 6.5.1) puede ser transfor-mada en la siguiente expresión lógicamente equivalente, aunque probablemente más eficiente

```
( ( VP WHERE P# = P# ( 'P2' ) ) JOIN V ) { PROVEEDOR }
```

(Ejercicio: ¿En qué sentido sería más eficiente la segunda expresión? ¿Por qué sólo "pobablemente"?).

---

*Puesto que (como hemos visto) el producto es un caso especial de junta, en esta lista de primil dríamos reemplazar producto por junta. Lo que es más, en realidad también necesitamos agregar RENAME a la lista, ya que nuestra álgebra (a diferencia de la definida en la referencia [6.1]) depende de la de denomi-nación de los atributos en lugar de la posición ordinal.

Entonces, el álgebra sirve como una base conveniente para la **optimization** (si necesita refrescar su memoria con respecto al concepto de optimization, consulte el capítulo 3, sección 3.5). Esto es, aun si el usuario declara la consulta en la primera de las dos formas mostradas arriba, el optimizador debe convertirla a la segunda forma antes de ejecutarla (de manera ideal, el rendimiento de una consulta determinada *no* debe depender de la forma específica en que el usuario la declara). Para más información, vea el capítulo 17.

Concluimos esta sección señalando que, precisamente debido a su naturaleza fundamental, a menudo se usa el álgebra como una clase de *norma* contra la cual se puede medir el poder expresivo de un lenguaje dado. En esencia, se dice que un lenguaje está **relacionalmente completo** [6.1] si es por lo menos tan poderoso como el álgebra; es decir, si sus expresiones permiten la definición de cada relación que puede ser definida por medio de expresiones del álgebra (es decir, del álgebra *original,* como describimos en las secciones anteriores). En el siguiente capítulo explicaremos con más detalle está noción de compleción relational.

6.7 OPERADORES ADICIONALES

Diversos escritores han propuesto nuevos operadores algebraicos desde que Codd definió sus ocho originales. En esta sección analizaremos con cierto detalle algunos de estos operadores: SEMIJOIN, SEMIMINUS, EXTEND, SUMMARIZE y TCLOSE. En términos de nuestra sintaxis de **Tutorial D,** estos operadores comprenden cinco nuevas formas de <*no proyectar*>, que específicamente son como sigue:

```
<semijuntar>
    ::=  <expresión relacional>
                    SEMIJOIN <expresión relacional>

<semidiferencia>
    ::=  <expresión relacional>
                    SEMIMINUS <expresión relacional>

<extender>
    ::=  EXTEND <expresión relacional>
                      ADD <lista de agregados de extender separados con comas>

<agregado de extender>
    ::=  <expresión> AS <nombre de atributo>


   ::= SUMMARIZE  <expresión  relacional>  PER  <expresión  relacional>
             ADD <lista de agregados de resumir separados con comas>

<agregado de resumir>
    ::=  <tipo de resumen> [ ( <expresión escalar> ) ]
                                      AS <nombre de atributo>

<tipo de resumen>
          COUNT    SUM       AVG | MAX | MIN | ALL | ANY
        | COUNTD   SUMD      AVGD

<cierret>
          TCLOSE <expresión relacional>
```

Las distintas <*expresiones relacionales*> mencionadas en las reglas de producción BNF anteriores no deben ser <*no proyectar*>.

Semi juntar

Sean A, *B, X, Y y* Z como están definidas en la subsección "Juntar", de la sección 6.4. Entonces, la semijunta de A con *B* (en ese orden), A SEMI JOIN *B,* está definida como equivalente a

```
( A JOIN B ) { X, Y }
```

En otras palabras, la semijunta de A con *B* es la junta de A y *B,* proyectada sobre los atributos de A. Entonces, el cuerpo del resultado es (a grandes rasgos) las tupias de *A* que tienen una contraparte en *B.*

Ejemplo: Obtener V#, PROVEEDOR, STATUS y CIUDAD de los proveedores que suministran la parte P2:

```
V SEMIJOIN ( VP WHERE P# = P# ( 'P2' ) )
```

Semidiferencia

De nuevo, sean A, *B, X, Y* y Z como están definidas en la subsección "Juntar", en la sección6.4. Entonces la **semidiferencia** entre A y *B* (en ese orden), A SEMIMINUS *B,* está definida conx equivalente a

```
A MINUS ( A SEMIJOIN B )
```

Entonces, el cuerpo del resultado es (a grandes rasgos) las tupias de *A* que no tienen contraparte en B.

Ejemplo: Obtener V#, PROVEEDOR, STATUS y CIUDAD de los proveedores que no suministran la parte P2:

```
V SEMIMINUS ( VP WHERE P# = P# ( ' P2' ) )
```

Extender

Tal vez haya notado que el álgebra, tal como la describimos hasta ahora, no tiene posibilidades de efectuar cálculos (como se entiende convencionalmente el término). Sin embargo, en la prác-tica esas posibilidades son obviamente necesarias. Por ejemplo, nos gustaría poder recuperar el valor de una expresión aritmética como PESO * 454 o referirnos a un valor como éste en la cáu-sula WHERE (recuerde que los pesos de las partes están dados en libras; la expresión PESO 454 convertirá dicho peso a gramos)*. La finalidad de la operación **extender** consiste en dar so-porte a dichas posibilidades. Para ser más precisos, EXTEND toma una relación y regresa otra que es idéntica a la primera, con excepción de que incluye un atributo adicional cuyos valores

Para efectos del ejemplo damos por hecho que "" es una operación válida entre pesos y enteros. ¿Cual es el tipo del resultado de esta operación?

se obtienen mediante la evaluación de cierta expresión de cálculo especificada. Por ejemplo, podríamos escribir:

```
EXTEND P ADD ( PESO * 454 ) AS PSGR
```

Esta expresión —por favor, observe que *se trata de* una expresión y no de un "comando" o instrucción, y por lo tanto puede ser anidada dentro de otras expresiones— produce una relación con el mismo encabezado de P, salvo que incluye además un atributo llamado PSGR. Cada tupia de la relación es la misma que la tupia correspondiente de P, con excepción de que incluye además un valor PSGR, calculado de acuerdo con la expresión aritmética especificada. Vea la figura 6.9.

p#	PARTE	COLOR	PESO	CIUDAD	PSGR
P1	Tuerca	Rojo	12.0	Londres	5448.0
P2	Perno	Verde	17.0	París	7718.0
P3	Tornillo	Azul	17.0	Roma	7718.0
P4	Tornillo	Rojo	14.0	Londres	6356.0
P5	Leva	Azul	12.0	París	5448.0
P6	Engrane	Rojo	19.0	Londres	8626.0

Figura 6.9 Ejemplo de EXTEND.

Importante: Observe que esta expresión EXTEND *no* ha modificado la varrel partes en la base de datos; es sólo una expresión, tal como lo es (por ejemplo) V JOIN VP y como cualquier otra expresión que produce simplemente un cierto resultado que en este caso en particular luce más bien como el valor actual de la varrel partes. (En otras palabras, EXTEND *no* es un equivalente del álgebra relacional para ALTER TABLE de SQL.)

Ahora podemos usar el atributo PSGR en proyecciones, restricciones, etcétera. Por ejemplo:

```
( ( EXTEND P ADD ( PESO * 454 ) AS PSGR )
          WHERE PSGR > PESO ( 10000.0 ) ) { ALL BUT PSGR }
```

Nota: Desde luego, un lenguaje más amigable con el usuario permitiría que la expresión de cálculo apareciera directamente en la cláusula WHERE, como aquí:

```
P WHERE ( PESO * 454 ) > PESO ( 10000.0 )
```

Sin embargo, dicha posibilidad es solamente un adorno sintáctico.

Entonces, en general el valor de la expresión

EXTEND *A* ADD *exp* AS *Z*

está definido como una relación (a) con un encabezado igual al encabezado de *A* extendido con el nuevo atributo Z y (b) con un cuerpo que consiste en todas las tupias f, tal que *t* es una tupia de *A* extendida con un valor para el nuevo atributo Z, el cual se calcula evaluando la expresión *exp* sobre esa tupia de A. La relación *A* no debe tener un atributo de nombre Z y *exp* no debe hacer

referencia a Z. Observe que el resultado tiene una cardinalidad igual a la de A y un grado igual al de *A* más uno. El tipo de Z en ese resultado es del tipo de *exp*. He aquí algunos ejemplos más:

1. `EXTEND V ADD 'Proveedor' AS ETIQ`

   Esta expresión en efecto etiqueta cada tupia del valor actual de la varrel V con el valor de la cadena de caracteres 'Proveedor' (una literal —o de manera más general, una invocación al selector— desde luego es un caso especial de una expresión de cálculo).

2. `EXTEND ( P JOIN VP ) ADD ( PESO * CANT ) AS PSENV`

   Este ejemplo ilustra la aplicación de EXTEND al resultado de una expresión relational que es más complicada que un simple nombre de varrel.

3. `( EXTEND V ADD CIUDAD AS CIUDADV ) { ALL BUT CIUDAD }`

   Un nombre de atributo como CIUDAD también es una expresión de cálculo válida. Observe que este ejemplo en particular es equivalente a

   ```
   V RENAME CIUDAD AS CIUDADV
   ```

   En otras palabras, RENAME no es primitiva; puede ser definida en términos de EXTEND (y proyectar). Por supuesto, por razones de uso, no querríamos descartar nuestro conocido operador RENAME, pero al menos es interesante notar que en realidad se trata sólo de una abreviatura.

4. `EXTEND P ADD PESO * 454 AS PSGR, PESO * 16 AS PSOZ`

   Este ejemplo ilustra un "EXTEND múltiple".

5. ```
   EXTEND V
   ADD COUNT ( ( VP RENAME V# AS X ) WHERE X = V# )
   AS NP
   ```

   El resultado de esta expresión lo muestra la figura 6.10. *Explicación:*

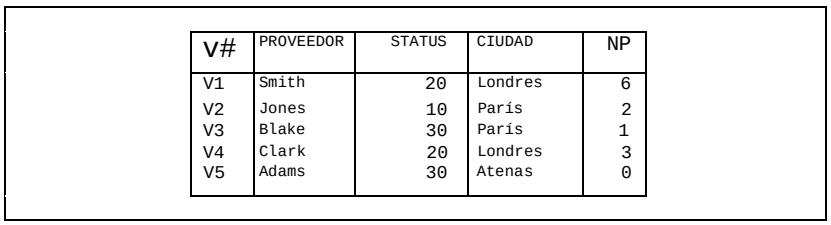
   - Para una tupia dada de proveedor en el valor actual de la varrel V, la expresión

     ```
     ( ( VP RENAME V# AS X ) WHERE X = V# )
     ```

     produce el conjunto de tupias de envíos correspondientes a esa tupia de proveedor en el valor actual de la varrel VP.
   - Entonces, el operador de totales COUNT se aplica a ese conjunto de tupias de envíos y regresa la cardinalidad correspondiente (que por supuesto es un valor escalar).

   Por lo tanto, el atributo NP en el resultado representa el número de partes suministradas el proveedor identificado por el valor V# correspondiente. En particular, observe el valor NP para el proveedor V5; el conjunto de tupias VP del proveedor V5 está vacío, por lo que la invocación a COUNT regresa cero.

   Ampliaremos un poco más esta cuestión de los operadores de totales. En general, finalidad de dichos operadores es derivar un solo valor escalar de los valores que apare-
     ```



V#	PROVEEDOR	STATUS	CIUDAD	NP
V1	Smith	20	Londres	6
V2	Jones	10	París	2
V3	Blake	30	París	1
V4	Clark	20	Londres	3
V5	Adams	30	Atenas	0

Figura **6.10** Otro ejemplo de EXTEND.

cen en algún atributo especificado de cierta relación especificada. Ejemplos típicos son COUNT, SUM, AVG, MAX, MIN, ALL y ANY. En **Tutorial D,** una <*invocación a operador de totales*> —la cual, debido a que regresa un valor escalar, es un caso especial de una <*expresión escalar*>— toma la forma general

```
<nombre operador> ( <expresión relacional> [, <nombre de atributo> ] )
```

Si el <*nombre operador*> es COUNT, el <*nombre de atributo*> es irrelevante y debe ser omitido; en caso contrario, puede ser omitido si y sólo si la <*expresión relacional*> denota una relación de grado uno, en cuyo caso el único atributo del resultado de esa <*expresión relacional*> se toma de manera predeterminada. Aquí tenemos un par de ejemplos:

```
SUM ( VP WHERE V# = V# ( 'V1' ), CANT )

SUM ( ( VP WHERE V# = V# ( 'V1' ) ) { CANT } )
```

Observe la diferencia entre estas dos expresiones; la primera da el total de las cantidades de envíos del proveedor VI, la segunda da el total de las cantidades de envíos *distintas* del proveedor V1.

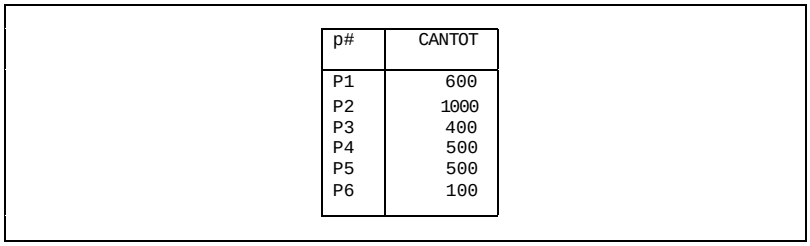
Si el argumento para un operador de totales es un conjunto vacío, COUNT (como hemos visto) regresa cero y lo mismo hace SUM; por su parte, MAX y MIN regresan el valor más bajo y el más alto (respectivamente) en el dominio relevante; ALL y ANY regresan *verdadero y falso,* respectivamente; y AVG genera una excepción.

Resumir

Debemos comenzar esta subsección diciendo que la versión de SUMMARIZE que se explicó aquí no es la misma que expliqué en ediciones anteriores de este libro; de hecho, es una versión mejorada que supera ciertas dificultades que surgieron con esa versión anterior con respecto a las relaciones vacías.

Como hemos visto, el operador *extender* proporciona una forma de incorporar cálculos "horizontales" o "en el nivel de fila" dentro del álgebra relacional. El operador **resumir** realiza la función análoga para los cálculos "verticales" o "en el nivel de columna". Por ejemplo, la expresión

```
SUMMARIZE VP PER P { P# } ADD SUM ( CANT ) AS CANTOT
```



p#	CANTOT
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

Figura 6.11 Un ejemplo de SUMMARIZE.

da como resultado una relación con encabezado {P#,CANTOT}, en la que hay una t cada valor P# de la proyección VP{P#} que contiene ese valor P# y la cantidad total correspondiente (vea la figura 6.11). En otras palabras, la relación VP se agrupa de manera conceptual en *conjuntos* de tupias, un conjunto para cada valor P# en P{P#}, y luego cada grupo es usado para generar una tupia de resultado.

En general, el valor de la expresión

```
SUMMARIZE A PER B ADD resumen AS Z
```

es definido como sigue:

- Primero, *B* debe ser del mismo tipo que alguna proyección de *A;* es decir, todo atributo *B* debe ser un atributo de *A.* Sean *Al, A2, ...,An* los atributos de esa proyección (de manera equivalente para *B).*
- El encabezado del resultado consiste en los atributos *A1, A2, ...,An* más el nuevo atributo
- El cuerpo del resultado consiste en todas las tupias *t* tal que t es una tupia de *B* extendida con un valor para el nuevo atributo Z. Ese nuevo valor Z se calcula evaluando *resumen dobre* todas las tupias de *A* que tengan los mismos valores *de Al, A2, ..., An* que la tupia *t.* (Desde luego, si ninguna tupia de *A* tiene los mismos valores de *Al, A2, ..., An* que la tupia t. re-*sumen* será evaluado sobre un conjunto vacío.) La relación *B* no debe tener un atril llamado Z y *resumen* no debe hacer referencia a Z Observe que el resultado tiene una cardinalidad igual a la de *B* y un grado igual al de *B* más uno. En ese resultado, el tipo de Z es del tipo de *resumen.*

Aquí tenemos otro ejemplo:

```
SUMMARIZE ( P JOIN VP ) PER P { CIUDAD } ADD COUNT AS NVP
```

El resultado luce como sigue:

CIUDAD	NVP
Londres	5
París	6
Roma	1

En otras palabras, el resultado contiene una tupia para cada una de las tres ciudades de partes (Londres, París y Roma), y muestra en cada caso la cantidad de envíos de partes almacenadas en esa ciudad.

Puntos a destacar:

1. Nuestra sintaxis permite "SUMMARIZES múltiples". Por ejemplo:

```
SUMMARIZE VP PER P { P# } ADD SUM ( CANT ) AS CANTOT
                         AVG ( CANT ) AS CANTPROM
```

2. La forma general de <*resumir*> es (de nuevo) como sigue:

```
SUMMARIZE <expresión relacional>
        PER <expresión relacional>
                ADD <lista de agregados de resumir separados con comas>
```

   A su vez, cada <*agregado de resumir*> toma la forma:

```
<tipo de resumen> [ ( <expresión escalar> ) ] AS <nombre de atributo>
```

   Los <*tipos de resumen* > generalmente válidos son COUNT, SUM, AVG, MAX, MIN, ALL, ANY, COUNTD, SUMD y AVGD. La "D" ("distinto") en COUNTD, SUMD y AVGD significa "elimina valores duplicados redundantes antes de realizar el resumen". La <*expresión escalar*> puede incluir referencias a atributos de la relación denotada por la <*expresión relacional*> que está inmediatamente después de la palabra reservada SUMMARIZE. *Nota:* La <*expresión escalar*> (y los paréntesis que la encierran) pueden ser omitidos sólo si el <*tipo de resumen*> es COUNT.

   Por cierto, observe que <*agregado de resumir*> *no* es lo mismo que una <*invocación a operador de totales*>. Una <*invocación a operador de totales*> es una expresión escalar y puede aparecer siempre que se permita una invocación al selector escalar (en particular, una literal escalar). Por el contrario, <*agregado de resumir*> es simplemente un operando de SUMMARIZE; no es una expresión escalar, no tiene significado fuera del contexto de SUMMARIZE y de hecho no puede aparecer fuera de ese contexto.

3. Como ya se habrá dado cuenta, SUMMARIZE no es un operador primitivo, puede ser simu lado por medio de EXTEND. Por ejemplo, la expresión

```
SUMMARIZE VP PER V { V# } ADD COUNT AS NP
```

   está definida como una abreviatura de lo siguiente:

```
( EXTEND V { V# }
         ADD ( ( VP RENAME V# AS X ) WHERE X = V# ) AS Y,
             COUNT ( Y ) AS NP ) {
V#, NP }
```

   O de manera equivalente a:

```
WITH ( V { V# } ) AS T1,
     ( VP RENAME V# AS X ) AS T2,
     ( EXTEND T1 ADD ( T2           V# ) AS Y ) AS T3,
     WHERE X                  Y ) AS NP ) AS T4
     ( EXTEND T3 ADD COUNT (
T4 { V#, NP }
```

4. Considere el siguiente ejemplo

```
SUMMARIZE VP PER VP { } ADD SUM ( CANT ) AS GRANTOTAL
```

En este ejemplo, el agrupamiento y el resumen son realizados "por" *una relación que no tiene atributos en absoluto.* Sea *vp* el valor actual de la varrel VP y supongamos por el momento esa relación *vp* sí contiene por lo menos una tupia. Entonces todas esas tupias *vp* tienen el mismo valor para los no atributos en absoluto, es decir, la 0-tupla [5.5]; por lo tanto, sólo hay un grupo y sólo una tupia en el resultado general. En otras palabras, el cálculo de totales se realiza samente una vez para toda la relación *vp.* Por lo tanto, SUMMARIZE da como resulta una relación con un atributo y una tupia; el atributo se llama GRANTOTAL y el único valor escal en la tupia resultante es el total de todos los valores de CANT de la relación original *vp.*

Si por el contrario, la relación original *vp* no tiene tupia alguna, entonces no hay grupos y por lo tanto no hay tupias de resultado; es decir, la relación de resultado tambi vacía. En contraste, la siguiente expresión*

```
SUMMARIZE VP PER RELATION { TUPLE { } }
            ADD SUM ( CANT ) AS GRANTOTAL
```

"funcionará" (es decir, regresará la respuesta "correcta"; es decir, cero) aun cuando vacía. Para ser más precisos, regresará una relación con un atributo denominado GRANTOTAL y una tupia que contiene el valor de GRANTOTAL igual a cero. Por lo tanto sugerimos que debe ser posible omitir en SUMMARIZE la cláusula PER, como aquí:

```
SUMMARIZE VP ADD SUM ( CANT ) AS GRANTOTAL
```

Omitir la cláusula PER equivaldría a especificar una cláusula PER de la forma

```
PER RELATION { TUPLE { } }
```

Cierret

"Cierret" es un acrónimo de *cierre transitivo (transitive closure).* Lo mencionamos aquí princi-pálmente para ofrecer una visión completa; la información detallada está fuera del alcance de este capítulo. Sin embargo, por lo menos definiremos la operación como sigue. Sea/ relación binaria con los atributos *X y Y,* ambos del mismo tipo *T;* entonces, el cierre transitivo de A (TCLOSE A) es una relación A^+ con un encabezado igual que el de A y con un supero» junto de A como cuerpo, definido como sigue. La tupia

{ x:x , y:y }

aparece en *A+* si y sólo si aparece en A o existe una secuencia de valores *zl,z2* ..... *zn,* todo del tipo *T,* tal que todas las tupias

X:x , Y:z1 *X:z1 , Y:z2* *{ X:zn , Y: y }*

*La expresión RELATION { TUPLE ( ) } en la cláusula PER de este ejemplo denota una relación -- de hecho, la *única* relación— sin atributo alguno pero con una tupia (la 0-tupla). Es posible abreviar a TABLE_DEE (véa las referencias [3.3], [5.5] y [6.2]).

aparecen en *A*. (En otras palabras, la tupia *"(x,y)"* aparece sólo si hay una ruta de acceso en el grafo que esté representado por la relación *A* del nodo *x* al nodo *y*, en términos generales. Observe que el cuerpo de *A+* incluye necesariamente el cuerpo de *A* como un subconjunto.) Para una mayor explicación de este tema consulte el capítulo 23.

6.8 AGRUPAMIENTO Y DESAGRUPAMIENTO

El hecho de que podamos tener relaciones con atributos cuyos valores son **a** su vez relaciones, conduce a la necesidad de ciertos operadores relacionales adicionales, los cuales llamamos *agrupar* y *desagrupar*, respectivamente [3.3]. Aquí tenemos primero un ejemplo de *agrupar:*

```
VP GROUP ( P#, CANT ) AS PC
```

Dados nuestros datos de muestra usuales, esta expresión produce el resultado que muestra la figura 6.12. *Nota:* Tal vez encuentre útil emplear esta figura para comprobar las explicaciones que siguen, ya que —lamentable pero inevitablemente— son un poco abstractas.

V#	PC
V1	

P#	CANT
P1	300
P2	200
P3	400
P4	200
P5	100
P6	100

V2

P#	CANT
P1	300
P2	400

V3

P#	CANT
P2	200

P#	CANT
P2	200
P4	300
P5	400

Figura 6.12 Agrupamiento de VP por V#.

Comenzaremos por observar que la expresión original

```
VP GROUP ( P#, CANT ) AS PC
```

podría ser leída como "agrupa VP por V#"; ya que V# es el único atributo de VP que *no* se menciona en la cláusula GROUP. El resultado es una relación definida como sigue. Primero, el encabezado luce como el siguiente:

```
{ V# V#, PC RELATION { P# P#, CANT CANT } }
```

En otras palabras, consiste en un atributo PC con valor de relación (donde PC tiene a su vez los atributos P# y CANT) junto con todos los demás atributos de VP (por supuesto, aquí "todos los demás atributos de VP" significa sólo el atributo V#). Segundo, el cuerpo contiene exactamente una tupia para cada valor distinto de V# en VP (y ninguna otra tupia). Cada tupia de ese cuerpo consiste en el valor V# aplicable (digamos, v) junto con un valor PC (digamos, pc) obtenido como sigue:

- Cada tupia VP es sustituida conceptualmente por una tupia (digamos, *x)* en la que los componentes P# y CANT han sido "envueltos" dentro de un componente con valor de tupia (digamos, *y).*
- Los componentes y de todas estas tupias *x* (en las que el valor V # es igual a v) se "agru pan" dentro de una relación *pe* y de ahí se genera una tupia con valor V# igual a v y valor PC igual *a. pc.*

Por lo tanto, el resultado general es como muestra la figura 6.12.

Pasemos ahora a ***desagrupar***. Sea VPC la relación que muestra la figura 6.12. Entonces la expresión

```
VPC UNGROUP PC
```

nos lleva de regreso (quizás sin sorpresa) a nuestra relación de muestra usual VP. Para ser más específicos, produce una relación definida de la siguiente manera. Primero, el encabezado luce como el siguiente:

```
{ V# V#,  P# P#,  CANT CANT }
```

En otras palabras, el encabezado consiste en los atributos P# y CANT (derivados del atrito PC), junto con todos los demás atributos de VPC (es decir, sólo el atributo V# en el ejemplol Segundo, el cuerpo contiene exactamente una tupia para cada combinación de una tupia de VPC con una tupia del valor de PC dentro de esa tupia (y ninguna otra tupia). Cada tupia de ese cuerpo consiste en el valor aplicable de V# (digamos v), junto con los valores P# y CANT (digamos p y c) obtenidos como sigue:

- Cada tupia VPC es sustituida en forma conceptual por un conjunto de tupias, cada una de
  esas tupias (digamos *x)* corresponde a cada tupia del valor de PC para esa tupia VPC. Cad una de estas tupias *x* contiene un componente V# (digamos v) igual al componente V# de la tupia VPC en cuestión, así como un componente con valor de tupia (digamos y) igual a alguna tupia del componente PC de la tupia VPC en cuestión.
- Los componentes *y* de cada una de estas tupias *x* (en las que el valor V# es igual a v) s "desenvueltos" en componentes independientes P# y CANT (digamos *p* y c), y por ello se genera una tupia de resultado con el valor V# igual a *v,* el valor P# igual a p y el valor CANT igual a *c*.

Por lo tanto, tal como afirmamos, el resultado general es nuestra relación usual VP de ejemplo. Como puede ver, GROUP y UNGROUP ofrecen en conjunto lo que por lo regular se conoce como posibilidades de "anidar" y "desanidar" relaciones. Sin embargo, preferimos la terminología agrupar/desagrupar, ya que anidar/desanidar se asocia mucho con el concepto de las relaciones NF^2, un concepto que encontramos de algún modo confuso, para comenzar, y que no apoyamos.

Para completar la visión, cerramos esta sección con algunas observaciones con respecto a la característica *reversible* de las operaciones GROUP y UNGROUP (aunque estamos conscientes de que nuestras observaciones podrían no ser del todo comprensibles en una primera lectura). Si de alguna forma agrupamos una cierta relación *r*, siempre existe un desagrupamiento inverso que nos lleve de vuelta a *r*. Sin embargo, si de alguna manera desagrupamos cierta relación *r*, el agrupamiento inverso que nos lleve de vuelta a **r** podría no existir. Aquí tenemos un ejemplo (basado un ejemplo de la referencia [5.4]). Suponga que comenzamos con la relación DOS (vea la figura 6.13) y la desagrupamos para obtener la relación TRES. Si ahora agrupamos TRES por A (y denominamos una vez más, RVX al atributo con valor de relación resultante), obtenemos no la relación DOS, sino la UNO.

Observe que, en UNO, RVX es (necesariamente) *funcionalmente dependiente* de A, el cual por lo tanto constituye una clave candidata (vea los capítulos 8 y 10). Si ahora desagrupamos UNO, regresaremos a TRES, y ya vimos que TRES puede agruparse para dar UNO; por lo que en efecto las relaciones agrupar y desagrupar son inversas para este par de relaciones en particular. En general, es la dependencia funcional la que es crucial para determinar si un desagrupamiento dado es reversible o no. De hecho, si la relación ***r*** tiene un atributo con valor de relación *RVX*, entonces ***r*** se puede desagrupar de manera reversible (con respecto a *RVX)* si y sólo si se cumplen las dos condiciones siguientes:

- Ninguna tupia de *r* tiene una relación vacía como su valor *RVX*.
- *RVX* es funcionalmente dependiente de la combinación de todos los demás atributos de *r*. Otra forma de decirlo es que debe haber alguna clave candidata de *r* que no incluya como componente a *RVX*.

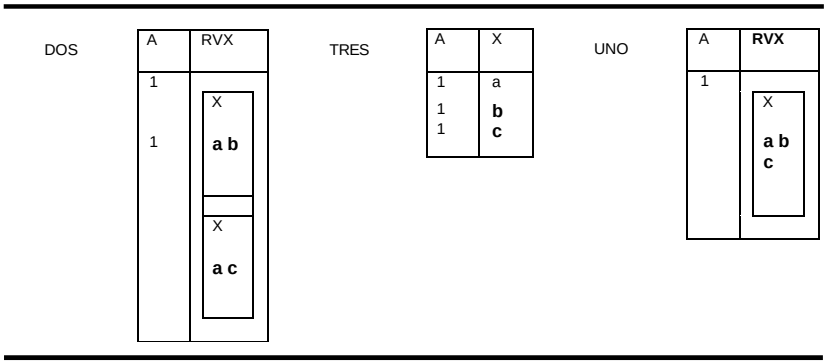


Figura 6.13 El desagrupamiento y el (re)agrupamiento no son necesariamente reversibles.

6.9 COMPARACIONES RELACIONALES

El álgebra relational, tal como se definió originalmente, no incluía ninguna forma directa comparar dos relaciones; por ejemplo, comprobar su igualdad o comprobar si cada tupia que aparecía en una relación lo hacía también en otra (es decir, comprobar si una relación era un subconjunto de otra, hablando en términos generales). Una consecuencia de esta omisión que ciertas consultas eran extremadamente difíciles de expresar (para un ejemplo, vea el eje-cicio 6.48 al final de este capítulo). Sin embargo, la omisión se repara con facilidad. En primer lugar, definimos una nueva clase de *condición* (una **comparación** relacional) con la siguiente sintaxis:

<expresión relacionala ç <expresión relacional>

Las relaciones denotadas por las dos <*expresiones relacionales*> deben ser del mi tipo. El operador de comparación 0 puede ser cualquiera de los siguientes:

= (igual a)

<> (desigual a)

< (subconjunto de)

< (subconjunto propio de)

> (superconjunto de)

> (superconjunto propio de)

Nota: La elección de los símbolos de los operadores tal vez no es la más adecuada, ya que (por ejemplo) la negación de "A es un subconjunto propio de *B"* ciertamente no es *"A* es un superconjunto de *B"* (es decir, "<" y ">" no son inversos entre sí). Sin embargo, para los fines de este libro nos quedaremos con esos símbolos por razones tipográficas:

Aquí tenemos entonces un par de ejemplos:

1 . V { CIUDAD } = P { CIUDAD }

Significa: ¿Es la proyección de proveedores sobre CIUDAD la misma que la proyección partes sobre CIUDAD?

2. v { v# } > VP { v# }

Significa (parafraseado en gran medida): ¿Existe algún proveedor que no suministre parle alguna?

A continuación nos permitimos utilizar este nuevo tipo de condición en los lugares apropi dos dentro de las expresiones relacionales. Por ejemplo:

V WHERE ((VP RENAME V# AS X) WHERE X = V#) { *P# }* = P { P# }

Esta expresión da como resultado una relación que contiene las tupias de los proveedores **que** suministran todas las partes. *Explicación:*

- Para un proveedor dado, la expresión

```
( ( VP RENAME V# AS X ) WHERE X = V# ) { P# }
```

  produce el conjunto de números de parte que suministra ese proveedor.

- Ese conjunto de números de parte es comparado después con el conjunto de *todos* los números de parte. Si ambos conjuntos son iguales, la tupia del proveedor correspondiente aparece en el resultado.

Por supuesto, ya sabemos cómo formular esta consulta particular en términos de DIVIDEBY:

```
V JOIN ( V { V# } DIVIDEBY P { P# } PER VP { V#, P# } )
```

Sin embargo, podría creer que tratar con la versión de la comparación relacional conceptualmente es más fácil. Pero hay una punto que debemos dejar en claro. Las comparaciones relacionales no son *condiciones de restricción,* tal como definimos el término en la sección 6.4, y ¡el ejemplo anterior que comprende dicha comparación no es una restricción genuina! Más bien, es una forma abreviada de algo como lo siguiente:

```
WITH ( EXTEND V
       ADD ( ( VP RENAME V# AS X ) WHERE X = V# ) { P# }
       AS A ) AS T1, (
     EXTEND T1
       ADD P { P# }
       AS B ) AS T2 :
T2 WHERE A = B
```

Aquí, A y B son atributos con valor de relación y después de todo, la expresión final T2 WHERE A = B ahora *es* una restricción genuina. *Nota:* Por cierto, de lo anterior se desprende que —por lo menos, hablando en forma conceptual— el soporte de comparaciones relacionales requiere del soporte de atributos con valor de relación.

Una comparación relacional específica que se necesita muy a menudo es una comprobación para ver si una relación dada está vacía (es decir, si no contiene tupias). Una vez más, parece que vale la pena presentar una forma abreviada. Por lo tanto, definimos un operador con valor de verdad de la forma

```
IS_EMPTY ( <expresión relacional> )
```

el cual regresa *verdadero* si la expresión denotada por <*expresión relacional*> está vacía y regresa *falso* en caso contrario.

Otro requerimiento común es poder comprobar si una tupia *t* dada aparece dentro de una relación *r* dada. Bastará la siguiente comparación relacional:

```
RELATION { t } < r
```

Sin embargo, la siguiente forma abreviada —que le resultará muy familiar si conoce SQL— es un poco más fácil para el usuario:

```
t IN r
```

Aquí, IN es en realidad el operador de pertenencia de conjunto, representado por lo regular como ∈.

6.10 RESUMEN

Hemos explicado el **álgebra reladonal.** Comenzamos por enfatizar de nuevo la importancia de la propiedad de **derre** y de las **expresiones relacionales anidadas,** y explicamos que si ibamos a tomar en serio el cierre, entonces necesitábamos un conjunto de reglas de inferencia de tipos **de relación** (y por supuesto, nuestra versión del álgebra incluye dichas reglas).

El álgebra original consistía de ocho operadores: el conjunto tradicional de operadores **unión, intersección, diferencia** y **producto** (todos ellos modificados en cierta forma para tomar en cuenta el hecho de que sus operandos son relaciones muy específicas y no conjuntos arbitrarios), así como los operadores relacionales especiales **restringir, proyectar, juntar** y **dividir. A** este conjunto original agregamos **RENAME, SEMIJOIN, SEMIMINUS, EXTEND** y **SUMMARIZE** (y también mencionamos **TCLOSE** y explicamos brevemente **GROUP** y **UNGR0UP).** Para algunos de estos operadores se requiere que los dos operandos de relación sean del mis tipo (concepto antes llamado "compatibles con la unión"). También señalamos que no todos estos operadores son **primitivos;** varios de ellos pueden ser definidos en términos de otros. Mostra-mos cómo combinar los operadores dentro de expresiones que sirven para una variedad de fines: **recuperación, actualización** y otros más. También explicamos brevemente la idea de **t formar** dichas expresiones con fines de **optimización** (aunque explicaremos esta **idea con** mucho más detalle en el capítulo 17). Y consideramos la posibilidad de utilizar un enfoque paso **a paso** para manejar las consultas complejas, empleando **WITH** para introducir nombres las expresiones. Por último, explicamos la idea de las **comparaciones relacionales,** las cuales de alguna manera hacen que ciertas clases de consultas sean más fáciles de expresar (por lo regular aquellas que de otra forma requerirían de DIVIDEBY).

EJERCICIOS

6.1 En el cuerpo de este capítulo afirmamos que unión, intersección, producto y junta (natural) son
tanto conmutativas como asociativas. Verifique estas afirmaciones.

6.2 Del conjunto original de ocho operadores de Codd, se pueden considerar primitivos a union
diferencia, producto, restringir y proyectar. En términos de esos operadores primitivos, dé las definiciones de junta natural, intersección y (¡más difícil aún!) dividir.

6.3 Considere la expresión *A* JOIN *B.* Si *A* y *B* tienen encabezados disjuntos (es decir, no tienen
atributos en común), esta expresión es equivalente a *A* TIMES *B.* Verifique esta afirmación.¿Cuál
sería la expresión equivalente si A y 5 tuvieran encabezados *idénticos?*

6.4 Demuestre que los cinco operadores primitivos mencionados en el ejercicio 6.2 son en primitivos, en el sentido de que ninguno de ellos puede ser expresado en términos de los otros cuatro.

6.5 En la aritmética ordinaria, la multiplicación y la división son operaciones inversas. ¿Son TIMES
y DIVIDEBY operaciones inversas en el álgebra relacional?

6.6 Dada la base de datos usual de proveedores y partes, ¿cuál es el valor de la expresión VJ VP JOIN P? *Advertencia:* aquí hay una trampa.

6.7 Sea *A* una relación de grado *n.* ¿Cuántas diferentes proyecciones de *A* existen?

6.8 En la aritmética ordinaria hay un número especial 1 con la propiedad de que para todos los
números *n*

1 = 1

Decimos que 1 es la **identidad** con respecto a la multiplicación. ¿Existe alguna relación que juegue un papel similar en el álgebra relacional? Si la hay ¿cuál es?

6.9 En la aritmética ordinaria hay otro número especial, 0, con la propiedad de que para todos los números *n*

¿Existe alguna relación que juegue un papel similar en el álgebra relacional? Si la hay ¿cuál es?

6.10 Investigue el efecto de las operaciones algebraicas expuestas en este capítulo sobre las rela ciones que son las respuestas a los dos ejercicios previos.

6.11 En la sección 6.2 dijimos que la propiedad de cierre relacional era importante por la misma razón que era importante el cierre aritmético. Sin embargo, en la aritmética existe una situación de sagradable cuando se rompe la propiedad de cierre; para ser más específicos, la división entre cero. ¿Existe una situación similar en el álgebra relacional?

6.12 Los operadores unión, intersección y juntar fueron definidos originalmente como operadores *diádicos* (es decir, cada uno de ellos toma exactamente dos operandos). Sin embargo, en este capítulo mostramos como podían extenderse sin ambigüedad para convertirse en operadores nádicos para un valor arbitrario de $n > 1$; por ejemplo, la expresión *A* UNION *B* UNION *C* podría ser considerada sin ambigüedad como la unión *triádica* de *A, B* y *C*. ¿Pero que sucede con $n = 1$? ¿O con $n = 0$?

Ejercicios de consultas

Los ejercicios restantes se basan en la base de datos de proveedores, partes y proyectos (vea la figura 4.5 en la sección "Ejercicios" del capítulo 4 y la respuesta al ejercicio 5.4 en el capítulo 5). En cada caso se le pedirá que escriba una expresión de álgebra relacional para la consulta indicada. (Como una variante interesante, podría ver primero algunas de las respuestas y determinar lo que la expresión dada significa en lenguaje natural). Por conveniencia, mostramos abajo (en bosquejo) la estructura de la base de datos:

```
V     { V#, PROVEEDOR, STATUS, CIUDAD }
      PRIMARY KEY { V# }
P     { P#, PARTE, COLOR, PESO, CIUDAD }
      PRIMARY KEY { P# }
Y     { Y#, PROYECTO, CIUDAD }
      PRIMARY KEY { Y# }
VPY   { V#, P#, Y#, CANT }
      PRIMARY KEY { V#, P#, Y# }
      FOREIGN KEY { V# } REFERENCES V
      FOREIGN KEY { P# } REFERENCES P
      FOREIGN KEY { Y# } REFERENCES Y
```

6.13 Obtener todos los detalles de todos los proyectos.

6.14 Obtener todos los detalles de todos los proyectos en Londres.

6.15 Obtener los números de los proveedores que suministran al proyecto Y1.

6.16 Obtener todos los envíos donde la cantidad está en el rango de 300 a 750 inclusive.

6.17 Obtener todas las combinaciones parte-color/parte-ciudad. *Nota:* A partir de este punto, el tér mino "todos(as)" se entenderá como "todos los representados actualmente en la base de datos", no "todos los posibles".

6.18 Obtener todos los triples número de proveedor/número de parte/número de proyecto, tales que el proveedor, parte y proyecto indicados estén coubicados (es decir, todos en la misma ciudad).

6.19 Obtener todos los triples número de proveedor/número de parte/número de proyecto, tale los proveedores, partes y proyectos indicados no estén todos coubicados.
6.20 Obtener todos los triples número de proveedor/número de parte/número de proyecto, tales que
ningún par de los proveedores, partes y proyectos indicados esté coubicado.
6.21 Obtener todos los detalles de las partes suministradas por un proveedor de Londres.
6.22 Obtener los números de las partes suministradas por un proveedor de Londres para un proyecto en Londres.
6.23 Obtener todos los pares de nombres de ciudad tales que un proveedor en la primera ciudad sur nistre a un proyecto en la segunda ciudad.
6.24 Obtener los números de las partes suministradas para cualquier proyecto por un proveedor la misma ciudad del proyecto.
6.25 Obtener los números de los proyectos que suministra por lo menos un proveedor que no est la misma ciudad.
6.26 Obtener todos los pares de números de parte tales que algún proveedor suministre ambas indicadas.
6.27 Obtener el número total de proyectos a los que suministra el proveedor V1.
6.28 Obtener la cantidad total de la parte Pl suministrada por el proveedor V1.
6.29 Para cada parte que sea suministrada a un proyecto, obtener el número de parte, el número de
proyecto y la cantidad total correspondiente.
6.30 Obtener los números de parte de las partes suministradas a algún proyecto en una cantidad
promedio superior a 350.
6.31 Obtener los nombres de los proyectos que suministra el proveedor V1.
6.32 Obtener los colores de las partes suministradas por el proveedor V1.
6.33 Obtener los números de las partes suministradas a cualquier proyecto en Londres.
6.34 Obtener los números de proyectos que usan al menos una parte disponible del proveedor V1
6.35 Obtener los números de proveedores que suministran al menos una de las partes sumistradas al menos por un proveedor que a su vez suministra por lo menos una parte roja.
6.36 Obtener los números de proveedor de los proveedores con un status menor al del proveedor
6.37 Obtener los números de los proyectos cuya ciudad es la primera en la lista alfabética de dichas
ciudades.
6.38 Obtener los números de los proyectos a los que se suministra la parte Pl en una cantidad
promedio superior a la cantidad más grande en que se suministra cualquier parte al proyecto Y1
6.39 Obtener los números de proveedor de los proveedores que suministran la parte Pl a algún
proyecto en una cantidad superior a la cantidad de envío promedio de la parte Pl para ese proyecto.
6.40 Obtener los números de los proyectos a los que ningún proveedor de Londres suministra una
parte roja.
6.41 Obtener los números de los proyectos suministrados en su totalidad por el proveedor V1.
6.42 Obtener los números de las partes suministradas a todos los proyectos en Londres.
6.43 Obtener los números de los proveedores que suministran la misma parte a todos los proyectos.
6.44 Obtener los números de los proyectos suministrados con al menos todas las partes dis del proveedor V1.
6.45 Obtener todas las ciudades en las que esté ubicado al menos un proveedor, parte o proyectos.
6.46 Obtener los números de parte de las partes suministradas ya sea por un proveedor de Londres
o a un proyecto en Londres.
6.47 Obtener los pares número de proveedor/número de parte tales que el proveedor indicado suministre la parte indicada.

6.48 Obtener todos los pares de números de proveedor, digamos *Vx* y Vy, tales que *Vx* y Vy sumi nistren exactamente el mismo conjunto de partes cada uno. (Agradezco este problema a Fatma Mili de la Universidad de Oakland, en Rochester, Michigan. Para simplificar, podría usar la base de datos origi nal de proveedores y partes, en lugar de la base de datos ampliada de proveedores, partes y proyectos).

6.49 Obtener una versión "agrupada" de todos los envíos que muestren, para cada par número de proveedor/número de parte, los números de proyecto correspondientes y las cantidades en forma de una relación binaria.

6.50 Obtener una versión "desagrupada" de la relación producida en el ejercicio 6.49.

REFERENCIAS Y BIBLIOGRAFÍA

6.1 E. F. Codd: "Relational Completeness of Data Base Sublanguages", en Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6.* Englewood Cliffs, N.J.: Prentice-Hall (1972).

Éste es el artículo en el que Codd definió por primera vez *formalmente* los operadores algebraicos originales (por supuesto, las definiciones aparecieron también en la referencia [5.1], pero eran en cierto modo menos formales o por lo menos incompletas).

Quizás un aspecto desafortunado del artículo es que da por hecho "por ser conveniente para la notación y la explicación" que los atributos de una relación tienen un orden de izquierda a derecha y de ahí que puedan identificarse por su posición ordinal (aunque Codd sí dice que "[deben] usarse nombres más que números de posición... al almacenar o clasificar realmente la información"; y por supuesto, antes había dicho algo muy parecido en la referencia [5.1]). Por lo tanto, el artículo no menciona un operador de atributos RENAME y no considera la cuestión de la inferencia del tipo del resultado. Tal vez como consecuencia de estas omisiones, en la actualidad todavía es posible dirigir las mismas críticas para muchas explicaciones escritas del álgebra, en los productos SQL actuales y (en un grado ligeramente menor) también en el estándar de SQL.

En el capítulo 7 aparecen comentarios adicionales a este artículo, en especial en la sección 7.4. *Nota:* La referencia [3.3] describe una clase de álgebra con un "conjunto reducido de instrucciones", denominada A, que permite la definición sistemática de operadores más poderosos en términos de combinaciones adaptables de un número de primitivos muy reducido. De hecho, la referencia [3.3] muestra que toda la funcionalidad del álgebra original de Codd puede lograrse con sólo dos primitivos llamados *remove* y *nor.*

6.2 Hugh Darwen (escribe como Andrew Warden): "Adventures in Relationland", en C. J. Date, *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Una serie de artículos breves que examina diversos aspectos del modelo relacional y de los DBMS relacionales en un estilo original, entretenido e informativo. Los artículos tienen los siguientes títulos:

1. The Naming of Columns
2. In Praise of Marriage
3. The Keys of the Kingdom
4. Chivalry
5. A Constant Friend
6. Table_Dee and Table_Dum
7. Into the Unknown

6.3 Hugh Darwen y C. J. Date: "Into the Great Divide", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).

Este artículo analiza (a) el operador original de Codd, dividir, tal como fue definido en la reí cia [6.1] y (b) una generalización de ese operador adjudicada a Hall, Hitchcock y Todd [6.10], a diferencia del operador original de Codd, permitía que cualquier relación fuera dividida **entre una** relación. (El operador original de Codd, dividir, fue definido sólo para relaciones dividendo)) visor tales que el encabezado del divisor fuera un subconjunto del encabezado del dividendo artículo muestra que ambos operadores se complican con las relaciones vacías, y esto da conx sultado que ninguno de ellos resuelva bien el problema para el que fue diseñado originalmente (es decir, ninguno de los dos llega a ser la contraparte del cuantificador universal que se pretend Para superar el problema, se proponen versiones revisadas de ambos operadores (la "División Pe-queña" y la "División Grande"). *Nota:* Como sugiere la sintaxis de **Tutorial D** para estos dos ope-radores, éstos son en realidad dos operadores diferentes; es decir, la División Grande no es (por desgracia) una extensión compatible hacia arriba de la División Pequeña. El artículo también giere que los operadores revisados ya no merecen llevar el nombre "dividir". Con respecto a última idea, vea el ejercicio 6.5.

Para fines de referencia, damos aquí una definición del operador original de Codd. dividir.Tengan las relaciones *A y B* encabezados *{X,Y} y {Y},* respectivamente (en donde X y *Y* pueden ser compuestos). Entonces la expresión *A* DIVIDEB Y *B* da una relación con el encabezado *{X} y un* cuerpo que consiste en todas las tupias *{X:x}* tales que una tupia *{X:x,Y:y}* aparece en *A* para todas las tupias *{Y:y}* que aparecen en *B.* En otras palabras, el resultado consiste a grandes rasgos chos valores *X* de *A* cuyos valores *Y* correspondientes (en *A)* incluyen todos los valores *Y*

6.4 C. J. Date: "Quota Queries" (en tres partes), en C. J. Date, Hugh Darwen y David y McGoveran,
Relational Database Writings 1994-1997. Reading, Mass.: Addison-Wesley (1998).

Una consulta de cuota *(quota query)* es una consulta que especifica un límite deseado s cardinalidad del resultado (por ejemplo la consulta "Obtener las tres partes más pesadas"). Aquí está una posible formulación de este ejemplo en **Tutorial D:**

```
P QUOTA ( 3, DESC PESO )
```

Esta expresión se define como una forma abreviada de lo siguiente:

```
( ( EXTEND P
   ADD COUNT ( ( P RENAME PESO AS PS ) WHERE PS > PESO ) AS
   MASPESADO ) WHERE MASPESADO < 3 ) { ALL BUT MASPESADO }
```

(donde los nombres PS y MASPESADO son arbitrarios.) Dados nuestros datos usuales de ejem-plo, el resultado consiste en las partes P2, P3 y P6.

El presente artículo de tres partes [6.4] analiza a profundidad los requerimientos de la consulta de cuota y propone varias formas sintácticas abreviadas para su manejo y f aspectos relacionados.

6.5 Michael J. Carey y Donald Kossmann: "On Saying 'Enough Already!' in SQL", Proc. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Otro artículo sobre consultas de cuota. A diferencia de la referencia [6.4], se concentra en aspec-tos de implementación más que aspectos de modelo. En las propuestas de este artículo, la c "Obtener las tres partes más pesadas" luciría como sigue:

```
SELECT *
FROM P
ORDER BY PESO DESC
STOP AFTER 3 ;
```

Un problema con este enfoque es que el STOP AFTER se aplica al resultado de (BY —que (como vimos en el capítulo 5, sección 5.3, subsección "Propiedades de las relacionales")

no es en absoluto una relación sino un arreglo o una lista—; de ahí que presuntamente el resultado general no es tampoco una relación y se viola la propiedad relacional de cierre. El artículo no aborda este aspecto.

Por supuesto, el resultado tal vez podría convertirse de nuevo en una relación, pero entonces caemos en otro problema: en general el resultado de STOP AFTER es impredecible. Por ejemplo, dados nuestros valores de ejemplo usuales, si en la consulta anterior de SQL reemplazamos STOP AFTER 3 por STOP AFTER 2, entonces el resultado no está bien definido.

La cláusula STOP AFTER ha sido implementada en un prototipo de investigación de IBM y podría por lo tanto abrirse camino en los productos de IBM y tal vez de ahí en el estándar de SQL (aunque esperamos que no sea antes de resolver satisfactoriamente los problemas identificados arriba).

6.6 R. C. Goldstein y A. J. Strnad: "The MacAIMS Data Management System", Proc. 1970 ACM SICFIDET Workshop on Data Description and Access (noviembre, 1970).

Vea el comentario de la referencia [6.7] que sigue.

6.7 A. J. Strnad: "The Relational Approach to the Management of Data Bases", Proc. IFIP Congress, Ljubljana, Yugoslavia (agosto, 1971).

Mencionamos a MacAIMS [6.6-6.7] principalmente por razones de interés histórico; parece haber sido el primer ejemplo de un sistema que manejara las relaciones n-arias y un lenguaje algebraico. Lo interesante al respecto es que se desarrolló en paralelo a, o por lo menos parcialmente independiente de, el trabajo de Codd sobre el modelo relacional. Sin embargo, a diferencia de este último, el esfuerzo de MacAIMS no condujo aparentemente a ninguna actividad subsiguiente importante.

6.8 M. G. Notley: "The Peterlee **IS/1** System", IBM UK Scientific Centre Report UKSC-0018 (marzo, 1972).

Vea los comentarios de la referencia [6.9].

6.9 S. J. P. Todd: "The Peterlee Relational Test Vehicle—A System Overview", *IBM Sys. J. 15*, No. 4(1976).

El PRTV (Peterlee Relational Test Vehicle) fue un sistema experimental desarrollado en el IBM UK Scientific Centre en Peterlee, Inglaterra. Se basó en un prototipo anterior —posiblemente la primera implementación de las ideas de Codd— denominado IS/1 [6.8]. Manejaba las relaciones n-arias y una versión del álgebra llamada ISBL (Lenguaje Base de Sistemas de Información), que se basaba en propuestas documentadas en la referencia [6.10]. Las ideas expuestas en el presente capítulo, con respecto a la inferencia de los tipos de relación, pueden ser rastreadas en el ISBL y en las propuestas de la referencia [6.10].

Los aspectos importantes del PRTV comprenden a los siguientes:

« Manejaba RENAME, EXTEND y SUMMARIZE.

- Incorporó algunas técnicas de transformación de expresiones sofisticadas (vea el capítulo 17).
- Incluyó una característica de evaluación diferida, la cual fue importante tanto para la optimización como para el soporte de vistas (vea la explicación de WITH en el cuerpo de este capítulo).
- Proporcionó la "extensión de funciones"; es decir, la capacidad del usuario para definir sus propios operadores.

6.10 P. A. V. Hall, P. Hitchcock y S. J. P. Todd: "An Algebra of Relations for Machine Compu tation", Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, Calif, (enero, 1975).

6.11 Patrick A. V. Hall: "Relational Algebras, Logic, and Functional Programming", Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (junio, 1984).

Presenta una interpretación de programación funcional del álgebra relacional con el objetivo —parafraseando el artículo— (a) ofrecer una base teórica para los tan mencionados "4GLs"(vea el capítulo 2) y (b) integrar lenguajes funcionales basados en la lógica (vea el capítulo 23 lenguajes relacionales, de modo que puedan compartir tecnología de implementatión. El autor afirma que mientras que la programación lógica y las bases de datos se han ido acercando ent sí por algún tiempo, al momento de escribir los lenguajes funcionales o aplicativos se ha hecho poco caso a los aspectos de las bases de datos. Por lo tanto, el artículo se presenta principalmente como una contribución hacia un *nuevo acercamiento* entre los dos últimos.

6.12 Anthony Klug: "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions", *JACM 29*, No. 3 (julio, 1982).

Define extensiones tanto para el álgebra relacional original como para el cálculo relational origi-nal (vea el capítulo 7), de manera que puedan soportar operadores de totales. También demuestra la equivalencia de los dos formalismos extendidos.

RESPUESTAS A EJERCICIOS SELECCIONADOS

Nota: Las respuestas dadas a los ejercicios 6.13 al 6.50 no son las únicas posibles. 6.2

JOIN lo expliqué en la sección 6.4. INTERSECT puede ser definido como sigue:

```
A INTERSECT B s A MINUS ( A MINUS S )
```

o (igualmente bien)

```
A INTERSECT B = B MINUS ( B MINUS A )
```

Estas equivalencias, aunque válidas, son ligeramente insatisfactorias, ya que *A* INTERSEC es simétrica en *A* y *B* y no lo es en las otras dos expresiones. En contraste, aquí tenemos un equivalente simétrico:

```
( A MINUS ( A MINUS B ) ) UNION ( S MINUS ( B MINUS A ) )
```

Nota: Dado que *A* y *B* deben ser del mismo tipo, también tenemos:

```
A INTERSECT B s A JOIN B
```

En lo que respecta a DIVIDEBY, tenemos:

```
A DIVIDEBY S PER C = A { X }
                     MINUS ( ( A { X } TIMES B { Y } )
                                  MINUS C { X, Y } ) { X }
```

Aquí, Xes el conjunto de atributos comunes *aA y C,* y Y es el conjunto de atributos comunes a B y C. *Nota:* DIVIDEBY como lo acabamos de definir es en realidad una generalización de I; definida en el cuerpo del capítulo —aunque sigue siendo una División Pequeña [6.3]—, ya que antes dimos por hecho que *A* no tenía atributos además de *X, B* no tenía atributos además de *Y,* yC no tenían atributos además de *X y Y.* La generalización precedente nos permitiría expresar, por ejemplo, lato sulta "Obtener los números de proveedor de los proveedores que suministran todas la partes" **de mena-**ra más sencilla únicamente como

```
V DIVIDEBY P PER VP
```

en lugar de expresarla como

```
V { V# } DIVIDEBY P { P# } PER VP { V#, P# }
```

6.3 *A* INTERSECT *B* (vea la respuesta del ejercicio 6.2). *Nota:* Observamos que ya que TIMES es un caso especial de JOIN, podríamos considerar a JOIN como un operador primitivo en lugar de TIMES; de hecho, precisamente porque es más general, resulta una mejor alternativa.

6.4 Damos un bosquejo informal *(muy* informal) de una "prueba" solamente.

- Producto es el único operador que puede incrementar el número de atributos, de modo que no puede ser simulado por ninguna combinación de los otros operadores. Por lo tanto es primitivo.
- Proyectar es el único operador que puede reducir el número de atributos, de modo que no puede ser simulado por ninguna combinación de los otros operadores. Por lo tanto es primitivo.
- Unión es el único operador que puede incrementar el número de tupias (además de producto) y producto por lo regular incrementa también el número de atributos. Sean *A* y *B* las dos relaciones a "unir". Observe que *A* y *B* deben ser del mismo tipo y que su unión tiene exactamente los mis mos atributos que cada una de ellas. Si formamos "el *producto* de *A* y *B"* renombrando (por decir algo) primero todos los atributos de *B* y luego aplicando el operador de producto y usando des pués proyección para reducir el conjunto de atributos del resultado a sólo el conjunto de atributos *de A,* simplemente regresamos de nuevo a la relación original *A:*

  (A TIMES e) { *todos los atributos de A* } = *A*

  (a menos que *B* esté vacía; ignoramos este caso por razones de simplicidad). Por lo tanto, es posible usar producto para simular unión, lo cual hace a este último primitivo.

- » Diferencia no se puede ser simulada por medio de unión (ya que unión nunca reduce el número de tupias) ni mediante producto (en forma similar, por lo general) ni por proyección (ya que regularmente, proyectar reduce el número de atributos). Ni tampoco puede ser simulada mediante restringir, ya que diferencia es sensible a los valores que aparecen en la segunda relación, mientras que restringir no puede serlo (en virtud de la naturaleza de una condición de restricción). Por lo tanto, el operador diferencia es primitivo.
- Restringir es el único operador que permite comparar valores de atributos entre sí. Por lo tanto, restringir es primitivo.

6.5 La respuesta breve es no. El DIVIDEBY original de Codd sí satisfizo la propiedad de que

```
( A TIMES S ) DIVIDEBY B = A
```

Sin embargo:

- El DIVIDEBY de Codd era un operador diádico; nuestro DIVIDEBY es triádico, de ahí que no haya posibilidad de que satisfaga una propiedad similar.
- En todo caso, incluso con el DIVIDEBY de Codd, dividiré entre *B* y luego formar el producto cartesiano del resultado con *B* producirá una relación que *podría* ser idéntica a *A,* pero es más probable que sea algún subconjunto propio de *A:*

  ```
  ( A DIVIDEBY B ) TIMES B < A
  ```

Por lo tanto, el DIVIDEBY de Codd es más parecido a la división entera en la aritmética ordinaria (es decir, ignora el residuo).

6.6 La trampa es que la junta involucra el atributo CIUDAD, así como los atributos V# y P#. El res- sultado luce como el siguiente:

V#	PROVEEDOR	STATUS	CIUDAD	P#	CANT	PARTE	COLOR	PESO
V1	Smith	20	Londres	P1	300	Tuerra	Rojo	12.0
V1	Smith	20	Londres	P4	200	Tornillo	Rojo	14.0
V1	Smith	20	Londres	P6	100	Engrane	Rojo	19.0
V2	Jones	10	París	P2	400	Perno	Verde	17.8
V3	Blake	30	París	P2	200	Perno	Verde	17.0
V4	Clark	20	Londres	P4	200	Tornillo	Rojo	14.0

6.7 2". Esta cuenta incluye la proyección *identidad* (es decir, la proyección sobre todos los **atributos**
n), lo que produce un resultado idéntico a la relación original *A*, así como la proyección *nula* (es decir,
la proyección sobre ningún atributo en absoluto), lo que produce TABLE__DUM si la relación origi-
nal *A* está vacía y TABLE_DEE si no [5.5].

6.8 Sí, dicha relación existe —es decir, TABLE_DEE. TABLE_DEE (DEE para abreviar)- es el equivalente de 1 con respecto a la multiplicación en la aritmética ordinaria, ya que

R TIMES DEE = DEE TIMES *R* = *R*

para todas las relaciones *R*. En otras palabras, DEE es la **identidad** con respecto a TIMES (y de ma-nera más general, con respecto a JOIN).

6.9 No existe ninguna relación que se comporte con respecto a TIMES de una forma que sea *exac tamente* análoga a la forma en que se comporta el 0 con respecto a la multiplicación. Sin embargo, comportamiento de TABLE_DUM (DUM para abreviar) es en cierto modo una reminiscencia del com portamiento del 0, ya que

R TIMES DUM DUM TIMES fl una relación vacía con el mismo encabezado que *R*

para todas las relaciones *R*.

6.10 Primero, observe que las únicas relaciones que son del mismo tipo que DEE y DUM son las mismas DEE y DUM. Tenemos:

UNION	DEE	DUM	INTERSECT	DEE	DUM	MINUS	DfcE	D
DEE	DEE	DEE	DEE	DEE	DUM	DEE	DUM	DE
DUM	DEE	DUM	DUM	DUM	DUM	DUM	DUM	E

En el caso de la diferencia, el primer operando se muestra a la izquierda y el segundo en la parte super-rior (por supuesto, para los otros operadores, los operandos son intercambiables). Observe qué ren cencías existen entre estas tablas y las tablas de verdad para OR, AND y AND NOT, respectivamente; desde luego, la semejanza no es una coincidencia.

Consideremos ahora restringir y proyectar, tenemos:

- Toda restricción de DEE produce DEE si la condición de restricción es *verdadera,* DUM si es *falsa*.
- Toda restricción de DUM produce DUM.
- La proyección de cualquier relación sobre ningún atributo produce DUM cuando la reelación ori-ginal está vacía; en caso contrario, produce DEE. En particular, la proyección de DEE o regresa su entrada (necesariamente sobre ningún atributo en absoluto).

Para extender y resumir, tenemos:

- Extender DEE o DUM para agregar un nuevo atributo produce una relación de grado uno y la misma cardinalidad que su entrada.
- Resumir DEE o DUM (necesariamente por ningún atributo en absoluto) produce una relación de grado uno y la misma cardinalidad que su entrada.

Nota: Omitimos la consideración de DIVIDEBY, SEMIJOIN y SEMIMINUS debido a que no son primitivos. TCLOSE es irrelevante (sólo se aplica a relaciones binarias). También omitimos la consideración de GROUP y UNGROUP por razones obvias.

6.11 ¡No!

6.12 Podemos definir razonablemente (y lo hacemos) la unión, la intersección, el producto o la junta de una sola relación *R* sólo como *R* misma. Por lo que toca al caso cero, sea *RT* algún tipo de relación. Entonces:

- La unión de ninguna relación en absoluto de tipo *RT* es la relación vacía de tipo *RT*. Observe que debe existir alguna forma de especificar *RT* como un operando para la unión.
- La intersección de ninguna relación en absoluto de tipo *RT* es la relación "universal" de tipo *RT*, donde por el término "relación universal de tipo *RT*" queremos decir aquella relación de tipo *RT* cuyo cuerpo contiene todas las tupias posibles que se apegan al encabezado de esa relación (una vez más, debe existir alguna forma de especificar *RT* como un operando de la operación). Observe que el término "relación universal" se usa con frecuencia en la literatura con un signifi cado muy diferente; por ejemplo, vea la referencia [12.19].
- El producto y la junta de ninguna relación en absoluto son ambos TABLE_DEE.

6.13
```
Y
```

6.14
```
Y WHERE CIUDAD = 'Londres'
```

6.15
```
( VPY WHERE Y# = Y# ( 'Y1' ) ) { V# }
```

6.16
```
VPY WHERE CANT > CANT ( 300 ) AND CANT < CANT ( 750 )
```

6.17
```
P { COLOR, CIUDAD }
```

6.18
```
( V JOIN P JOIN Y ) { V#, P#, Y# }
```

6.19
```
( ( ( V RENAME CIUDAD AS CIUDADV ) TIMES
    ( P RENAME CIUDAD AS CIUDADP ) TIMES
    ( Y RENAME CIUDAD AS CIUDADY ) ) WHERE
CIUDADV * CIUDADP OR   CIUDADP <> CIUDAOY
OR   CIUDADY <> CIUDADV ) { V#, P#, Y# }
```

6.20
```
( ( ( V RENAME CIUDAD AS CIUDADV ) TIMES
    ( P RENAME CIUDAD AS CIUDADP ) TIMES
    ( Y RENAME CIUDAD AS CIUDADY ) ) WHERE
CIUDADV <> CIUDADP AND  CIUDADP <> CIUDADY
AND  CIUDADY * CIUDADV ) { V#, P#, Y# }
```

6.21
```
P SEMIJOIN ( VPY SEMIJOIN ( V WHERE CIUDAD = 'Londres' ) )
```

6.22 Sólo para recordarle la posibilidad, mostramos una solución paso a paso para este ejercicio:

```
WITH ( V WHERE CIUDAD - 'Londres' ) AS T1,
     ( Y WHERE CIUDAD = 'Londres' ) AS T2,
     ( VPY JOIN T1 ) AS T3,
     T3 { P#,
     T3 { P#, Y# } AS T4, (
     T4 JOIN T2 ) AS T5 : T5
     { P# }
```

Aquí está la misma consulta sin utilizar WITH:

```
( ( VPY JOIN ( V WHERE CIUDAD = 'Londres' ) ) { P#, Y# }
                JOIN ( Y WHERE CIUDAD = 'Londres' ) ) { P# }
```

Para los ejercicios restantes daremos una mezcla de soluciones (algunas usando WITH y otras no)

6.23
```
( ( V RENAME CIUDAD AS CIUDADV ) JOIN VPY JOIN
  ( Y RENAME CIUDAD AS CIUDADY ) ) { CIUDADV, CIUDADY }
```

6.24
```
( Y JOIN VPY JOIN V ) { P# }
```

6.25
```
( ( ( Y RENAME CIUDAD AS CIUDADY ) JOIN VPY JOIN
    ( V RENAME CIUDAD AS CIUDADV ) )
                        WHERE CIUDADY <> CIUDADV ) { Y# }
```

6.26
```
WITH ( VPY { V#, P# } RENAME V# AS XV#, P# AS XP# ) AS T1,
     ( VPY { V#, P# } RENAME V# AS YV#, P# AS YP# ) AS T2,
     ( T1 TIMES T2 ) AS T3,
     ( T3 WHERE XV# = YV# AND XP# < YP# ) AS T4 :
     T4 { XP#, YP# }
```

6.27
```
( SUMMARIZE VPY { V#, Y# }
  PER RELATION { TUPLE { V# V# ( 'V1' ) } }
  ADD COUNT AS N ) { N }
```

Le recordamos que aquí la expresión en la cláusula PER es una invocación al selector de relación hecho, es una literal de relación).

6.28
```
( SUMMARIZE VPY { V#, P#, CANT }
PER RELATION { TUPLE { V# V# ( ' V1 ' ), P# P# ( ' P1 ' ) } }
ADD SUM ( CANT ) AS C ) { C }
```

6.29
```
SUMMARIZE VPY PER VPY { P#, Y# } ADD SUM ( CANT ) AS C
```

6.30
```
WITH ( SUMMARIZE VPY PER VPY { P#, Y# }
                     ADD AVG ( CANT ) AS C ) AS T1, (
     T1 WHERE C > CANT ( 350 ) ) AS T2 : T2 { P# }
```

6.31
```
( Y JOIN ( VPY WHERE V# = V# ( 'V1' ) ) ) { PROYECTO }
```

6.32
```
( P JOIN ( VPY WHERE V# = V# ( 'V1' ) ) ) { COLOR }
```

6.33
```
( VPY JOIN ( Y WHERE CIUDAD = 'Londres' ) ) { P# }
```

6.34
```
( VPY JOIN ( VPY WHERE V# = V# ( 'V1' ) ) { P# } ) { Y# }
```

6.35
```
( ( ( VPY JOIN
      ( P WHERE COLOR = COLOR ( 'Rojo' ) ) { P# } ) { V# }
                        JOIN VPY ) { P# } JOIN VPY ) { V# }
```

```
6.36 WITH ( V { V#, STATUS } RENAME V# AS XV#,
                                   STATUS AS XSTATUS ) AS T1 ,
          ( V { V#, STATUS } RENAME V# AS YV#,
                                   STATUS AS YSTATUS ) AS T2, (
          T1 TIMES T2 ) AS T3, ( T3 WHERE XV# = V# ( 'V1' ) AND
                    XSTATUS > YSTATUS ) AS T4 :
          T4 { YV# }

6.37 ( ( EXTEND Y ADD MIN ( Y, CIUDAD ) AS PRIMERA )

                                          WHERE CIUDAD = PRIMERA ) { Y# }
```

¿Qué regresa esta consulta si la varrel Y está vacía?

```
6.38 WITH ( VPY RENAME Y# AS ZY# ) AS T1 ,
          ( T1 WHERE ZY# = Y# AND P# = P# ( 'P1' ) ) AS T2,
          ( VPY WHERE P# = P# ( ' P1 ' ) ) AS T3,
          ( EXTEND T3 ADD AVG ( T2, CANT ) AS CX ) AS T4,
          T4 { Y#, CX } AS T5,
          ( VPY WHERE Y# = Y# ( ' Y1 ' ) ) AS T6,
          ( EXTEND T6 ADD MAX ( T6, CANT ) AS CY ) AS T7,
          ( T5 TIMES T7 { CY } ) AS T8,
          ( T8 WHERE CX > CY ) AS T9 :
          T9 { Y# }

6.39 WITH ( VPY WHERE P# = P# ( ' P1 ' ) ) AS T1 ,
          T1 { V#, Y#, CANT } AS T2,
          ( T2 RENAME Y# AS XY#, CANT AS XC ) AS T3,
          ( SUMMARIZE T1 PER VPY { Y# }
                       ADD AVG ( CANT ) AS C ) AS T4, (
          T3 TIMES T4 ) AS T5,
          ( T5 WHERE XY# ■ Y# AND XC > C ) AS T6 :
          T6 { V# }

6.40 WITH ( V WHERE CIUDAD ■ 'Londres' ) { V# } AS T1 ,
          ( P WHERE COLOR ■ COLOR ( 'Rojo' ) ) AS T2,
          ( T1 JOIN VPY JOIN T2 ) AS T3 : Y { Y# }
          MINUS T3 { Y# }

6.41 Y { Y# } MINUS ( VPY WHERE V# <> V# ( 'V1 ' ) ) { Y# }

6.42 WITH ( ( VPY RENAME  P# AS X ) WHERE X = P# ) { Y# } AS T1 ,
          ( Y WHERE CIUDAD.= 'Londres' ) { Y# } AS T2,
          ( P WHERE T1 > T2 ) AS T3 : T3 { P# }

6.43 V { V#, P# } DIVIDEBY Y { Y# } PER VPY { V#, P#, Y# }

6.44 ( Y WHERE
         ( ( VPY RENAME Y# AS K ) WHERE K = Y# ) { P# } >
         ( VPY WHERE V# = V# ( 'V1' ) ) { P# } ) { Y# }

6.45 V { CIUDAD } UNION P { CIUDAD } UNION Y { CIUDAD }

6.46 ( VPY JOIN ( V WHERE CIUDAD = 'Londres' ) ) { P# }
      UNION ( VPY JOIN ( Y WHERE CIUDAD = 'Londres' ) ) {
     P# }

6.47 ( V TIMES P ) { V#, P# } MINUS VPY { V#, P# }
```

6.48 Mostramos dos soluciones a este problema. La primera, la cual debemos a Hugh Darwen, una sólo los operadores de las secciones 6.2 a 6.3:

```
WITH ( VP RENAME V# AS VA ) { VA, P# } AS T1 ,
     /• T1 {VA,P#} : VA suministra la parte P# */

     ( VP RENAME V# AS VB ) { VB, P# } AS T2,
     /* T2 {VB,P#} : VB suministra la parte P# */

     T1 { VA } AS T3,
     /* T3 {VA} : VA suministra alguna parte */

     T2 { VB } AS T4,
     /* T4 {VB} : VB suministra alguna parte */

     ( T1 TIMES T4 ) AS T5,
     /* T5 {VA,VB,P#} : VA suministra alguna parte y
                        VB suministra la parte P#   */

     ( T2 TIMES T3 ) AS T6,
     /* T6 {VA,VB,P#} : VB suministra alguna parte y
                        VA suministra la parte P#   */

     ( T1 JOIN T2 ) AS T7,
     /* T7 {VA,VB,P#} : VA y VB suministran ambos la parte P# */

     ( T3 TIMES T4 ) AS T8,
     /* T8 {VA,VB} : VA suministra alguna parte y
                     VB suministra alguna parte  */

     VP { P# } AS T9,
     /* T9 {P#} : la parte P# es suministrada por algún proveedor */

     ( T8 TIMES T9 ) AS T10,
     /* T10 {VA,VB,P#} :
        VA suministra alguna parte,
        VB suministra alguna parte y
        la parte P# es suministrada por algún proveedor */

     ( T10 MINUS T7 ) AS T11,
     /* T11 {VA,VB,P#} : la parte P# es suministrada
                         pero no por VA y VB        */

     ( T6 INTERSECT T11 ) AS T12,
     /* T12 {VA,VB,P#} : la parte P# es suministrada
                         por VA pero no por VB      */

     ( T5 INTERSECT T11 ) AS T13,
     /* T13 {VA,VB,P#} : la parte P# es suministrada
                         por VB pero no por VA      */

     T12 { VA, VB } AS T14, /*
     T14 {VA,VB} :
        VA suministra alguna parte que no suministra VB */

     T13 { VA, VB } AS T15,
     /* T15 {VA,VB} :
        VB suministra alguna parte que no suministra VA */

     ( T14 UNION T15 ) AS T16,
     /* T16 {VA,VB} : alguna parte es suministrada por VA
                      o por VB pero no por ambos       •/
```

```
     T7 { VA, VB } AS T17,
     /* T17 {VA.VB} :
        alguna parte es suministrada tanto por VA como por VB */

     ( T17 MINUS T16 ) AS T18,
     /* T18 {VA.VB} :
        alguna parte es suministrada tanto por VA como por VB y ninguna
        parte suministrada por VA no es suministrada por VB y ninguna
        parte suministrada por VB no es suministrada por VA — asi que VA y
        VB suministran exactamente las mismas partes */

     ( T18 WHERE VA < VB ) AS T19 :
     /* paso para ordenar */

T19
```

La segunda solución —¡que es mucho más directa!— hace uso de las comparaciones relacionales que presentamos en la sección 6.9.

```
WITH ( V RENAME V# AS VA ) { VA } AS RA ,
     ( V RENAME V# AS VB ) { VB } AS RB :
     ( RA TIMES RB )
         WHERE ( VP WHERE V# = VA ) { P# } =
               ( VP WHERE V# = VB ) { P# }
         AND  VA < VB
```

```
6.49 VPY GROUP ( Y#, CANT ) AS CY
```

6.50 Sea VPC la relación que denote el resultado de la expresión mostrada en la respuesta al ejercicio 6.49. Entonces:

```
  VPC UNGROUP CY
```

CAPITULO 7

Cálculo relacional

7.1 INTRODUCCIÓN

En el capítulo 6 dijimos que la parte de manipulación del modelo relacional estaba basada en el álgebra relacional; sin embargo, de igual forma pudimos haber dicho que estaba basada en el *cálcu-lo* relacional. En otras palabras, el álgebra y el cálculo son alternativos entre sí. La diferenc principal entre ellos es la siguiente: Mientras que el álgebra proporciona un conjunto de operado res explícitos —juntar, unión, proyectar, etcétera— que pueden usarse para indicar al sistema cómo *construir* cierta relación deseada a partir de relaciones dadas, el cálculo simplemente propor-ciona una notación para establecer la *definición* de esa relación deseada en términos de dicto relaciones dadas. Por ejemplo, considere la consulta "obtener los números de proveedor y ciudades para los proveedores que suministran la parte P2". Una formulación algebraica de esa con-sulta podría especificar operaciones como las siguientes (deliberadamente no empleamos la sintaxis formal del capítulo 6):

- Primero, juntar las tupias de proveedores y envíos sobre V#;
- A continuación, restringir el resultado de esa junta a las tupias de la parte P2;
- Por último, proyectar el resultado de esa restricción sobre V# y CIUDAD.

En contraste, una formulación del cálculo podría declarar simplemente:

- Obtener V# y CIUDAD de los proveedores tales que exista un envío VP con el mismo valor
  V# y con el valor P2 de P#.

En esta última formulación, el usuario simplemente declara las características de definición resultado deseado y deja al sistema que decida exactamente qué juntas, restricciones, etcétera. debe ejecutar a fin de construir el resultado.

Por lo tanto, podríamos decir que —por lo menos superficialmente— la formulación cálculo es *descriptiva,* mientras que la algebraica es *prescriptiva:* El cálculo simplemente cribe cuál *es* el problema, él álgebra prescribe un procedimiento para *resolver* ese problema. de manera *muy* informal: El álgebra es de procedimientos (aunque de alto nivel, siguen diendo procedimientos); el cálculo no lo es.

Sin embargo, subrayamos la idea de que las distinciones anteriores son sólo superficiales. El hecho es que *el álgebra y el cálculo son lógicamente equivalentes.* Para cada exprésión al-gebraica existe una equivalente del cálculo, y para cada expresión del cálculo existe una alge-braica equivalente. Existe una correspondencia uno a uno entre ambos. Por lo tanto, la diferencia entre ellos es en realidad una diferencia de *estilo.* Hay quienes dicen que el cálculo está más

cercano al lenguaje natural y el álgebra es tal vez más como un lenguaje de programación. Pero, nuevamente tales diferencias son más aparentes que reales; en particular, ninguno de los enfoques está genuinamente más cargado a los no procedimientos que el otro. En la sección 7.4 examinaremos en detalle esta cuestión de equivalencia.

El cálculo relacional está basado en una rama de la lógica matemática denominada cálculo de predicados. La idea de usar el cálculo de predicados como la base para un lenguaje de consulta parece tener su origen en un documento de Kuhns [7.6]. El concepto de cálculo relacional —es decir, un cálculo de predicados aplicado específicamente a las bases de datos relacionales— fue propuesto por primera vez por Codd en la referencia [6.1]; un lenguaje basado explícitamente en ese cálculo, y llamado *Sublenguaje de datos ALPHA,* también fue presentado por Codd en otro artículo [7.1]. El propio ALPHA no llegó a ser implementado, pero un lenguaje llamado *QUEL [7.5, 7.10-7.12]* —que ciertamente fue implementado y fue por algún tiempo un serio competidor de SQL— era muy parecido a éste; de hecho, el diseño de QUEL tuvo mucha influencia de ALPHA.

Las variables de alcance son una característica fundamental del cálculo. Para resumir, una variable de *alcance es aquella* que *"abarca" a alguna relación especificada;* es decir, una variable cuyos valores permitidos son tupías de esa relación. Por lo tanto, si la variable de alcance *V abarca* a la relación *r,* entonces, en cualquier momento dado, la expresión *"V"* denota alguna tupía t de r-. Por ejemplo, la consulta "Obtener los números de proveedor de los proveedores en Londres" podría expresarse en QUEL como sigue:

```
RANGE OF VX IS V ;
RETRIEVE ( VX.V0 ) WHERE VX.CIUDAD ■ "Londres" ;
```

Aquí, la variable de alcance es VX y *abarca* cualquier relación que sea el valor actual de la varreí V (la instrucción RANGE es una *definición* de esa variable de alcance). Por lo tanto, Ja instrucción RETRIEVE puede ser parafraseada como: "Para cada valor posible de Ja variable VX, recupera el componente V# de ese valor si y sólo si el componente CIUDAD tiene el valor "Londres".

Debido a su dependencia sobre las variables de alcance cuyos valores son tupias (y para distinguirlo del cálculo de *dominios* —vea el siguiente párrafo), el cálculo relacional original se conoce ahora como cálculo de **tupias.** En la sección 7.2, describo en detalle el cálculo de tupias. *Nota:* Por razones de simplicidad, a lo largo de este libro adoptamos la convención de que los términos *cálculo* y *cálculo relacional,* sin un calificador de "tupias" o "dominios", se refieren específicamente al cálculo de tupias (en donde represente una diferencia).

En la referencia [7.7], Lacroix y Pirotte propusieron una versión alternativa del cálculo denomidada cálculo de **dominios,** en la cuaJ Jas variables de alcance abarcan dominios en lugar le relaciones.* En la literatura se han propuesto diversos lenguajes de cálculo de dominios; proablemente el más conocido sea QBE (Consulta por Ejemplos) [7.14] (aunque QBE es en readad una especie de híbrido, también incorpora eJementos deJ cálculo de tupias). Existen varias implementaciones comerciaies de QBE. En la sección 7.6 esbozamos el cálculo de dominios y la anotación a la referencia [7.14], explico brevemente el QBE.

---

terminología es ilógica. Si se llama así al cálculo de dominios por la razón dada, entonces el cálculo de as debe por derecho ser llamado cálculo relacional.

Nota: Omitimos deliberadamente la explicación de los equivalentes del cálculo en ciertos temas del capítulo 6; es decir, el cierre transitivo, el agrupamiento y desagrupamiento. así como las comparaciones relacionales. También omitimos las versiones del cálculo para los operad relacionales de actualización. En la referencia [3.3], usted puede encontrar una breve explicí de dichos temas.

7.2 CALCULO DE TUPLAS

Así como en nuestras explicaciones del álgebra expuestas en el capítulo 6, primero presentamos una sintaxis concreta —tomando como patrón, aunque en forma deliberada y no exactamete idéntica, la versión del cálculo de **Tutorial D** definida en la referencia [3.3]— y después tinuamos con la explicación de la semántica. La subsección inmediata siguiente expone taxis, las subsecciones restantes consideran la semántica.

Sintaxis

Nota: Muchas de las reglas de sintaxis dadas en forma de prosa en esta subsecció no tendrán mucho sentido hasta que haya estudiado parte del material semántico que se present, lante, pero las reunimos aquí para fines de referencia posterior.

Es conveniente comenzar repitiendo la sintaxis de las <*expresiones relacionales*> *del* capítulo 6:

```
<expresión relaclonal>
     ::=  RELATION { <lista de expresiones de tupia separadas con comas> }
           <nombre de varrel> <operación relacional> ( <expresión
           relacional> )
```

En otras palabras, la sintaxis de las <*expresiones relacionales*> es la misma que antes, pero uno de los casos más importantes, la <*operación relacional*>, tiene ahora una definición muy diferente (vea más adelante).

```
<definición de variable de alcance>
    ::=     RANGEVAR <nombre de variable de alcance>
            RANGES OVER <lista de expresiones  relacionales separadas con comas> ;
```

Un <*nombre de variable de alcance*> puede ser usado como una <*expresión de tuplas*> pero sólo en ciertos contextos; para ser más precisos:

- Antes del calificador de punto en una <*referencia a atributo de alcance*>;
- Inmediatamente después del cuantificador en una <*expresión lógica cuantificada*>;
- Como un operando dentro de una <*expresión lógica*>;
- Como una <*prototupla*> o como (un operando dentro de) una <*expresión*> dentro de una <*prototupla*>.

*Como en el capítulo 6, no detallamos aquí las <*expresiones de tupla*>, confiando en que los ejf muestren ser suficientes para ilustrar la idea general.

```
<referencia a atributo de alcance>
    ::=    <nombre de variable de alcance>  .  <referencia a atributo>
                                              [ AS <nombre de atributo>  ]
```

Una <*referencia a atributo de alcance*> puede usarse como una <*expresión*>, pero sólo en ciertos contextos; para ser más específicos:

- Como un operando dentro de una <*expresión lógica*>;
- Como una <*prototupla*> o como (un operando dentro de) una <*expresión*> dentro de una <*prototupla*>.

```
<expresión lógica>
    ::=    ...  todas las posibilidades usuales,  junto con: |
           <expresión lógica cuantificada>
```

Las referencias a variables de alcance dentro de una <*expresión lógica*> pueden ser libres dentro de esa <*expresión lógica*> sólo si se cumplen los dos puntos siguientes:

- La <*expresión lógica*> aparece inmediatamente dentro de una <*operación relacional*> (es decir, la <*expresión lógica*> está inmediatamente después de la cláusula WHERE), y
- Aparece de inmediato una referencia (necesariamente libre) para esa variable de alcance exacta dentro de la <*prototupla*> que está contenida inmediatamente dentro de esa misma <*operación relacional*> (es decir, la <*prototupla*> que antecede inmediatamente a la pala bra clave WHERE).

Un aspecto de la terminología: En el contexto del cálculo relacional (versión de tupias o de dominios), a las expresiones lógicas a menudo se les denomina **fórmulas bien formadas** o WFFs. Nosotros mismos usaremos esta terminología en gran parte de lo que sigue.

```
<expresión lógica cuantificada>
    ::=    EXISTS <nombre de variable de alcance>  (  <expresión lógica>  ) |
           FORALL <nombre de variable de alcance>  (  <expresión lógica>  )

<operación relacional>
    ::=    <prototupla>  [ WHERE <expresión lógica>  ]
```

Así como en el álgebra del capítulo 6, una <*operación relacional*> es una forma de <*expresión relacional*>, pero (como señalamos antes) aquí le estamos dando una definición diferente. Las otras formas de <*expresión relacional*> —básicamente nombres de variables de relación e invocaciones al selector de relación— siguen siendo válidas, como antes.

```
<prototupla>
    ::=    <expresión de tupla>
```

Todas las referencias a variables de alcance que aparecen inmediatamente dentro de una <*prototupla*> deben estar libres dentro de esa <*prototupla*>. *Nota:* "Prototupla" se usa por "tupia prototipo"; el término es conveniente aunque no estándar.

Variables de alcance

Aquí tenemos algunos ejemplos de definiciones de variable de alcance (como de costumbre, en términos de proveedores y partes):

```
RANGEVAR VX RANGES OVER V ;
RANGEVAR VY RANGES OVER V ;
RANGEVAR VPX RANGES OVER VP ;
RANGEVAR VPY RANGES OVER VP ;
RANGEVAR PX RANGES OVER P ;

RANGEVAR VU RANGES OVER
       ( VX WHERE VX.CIUDAD = 'Londres' ) ,
       ( VX WHERE EXISTS VPX ( VPX.V# = VX.V# AND
                               VPX.P# = P# ( 'P1' ) ) ) ;
```

En este último ejemplo, la variable de alcance VU se define para abarcar la *unión* del con junto de tupias de aquellos proveedores que están ubicados en Londres y el conjunto de tuplas de aquellos proveedores que suministran la parte Pl. Observe que la definición de la variabl alcance VU hace uso de las variables de alcance VX y VPX. Observe también que en estas definiciones al "estilo unión", las relaciones a "unir" deben ser (por supuesto) todas del mismo **tipo.**

Nota: Las variables de alcance no son variables en el sentido usual de un lenguaje de pro-gramación, son variables en el sentido de la lógica. De hecho, de alguna manera son similares a los *indicadores de posición* o *parámetros* de predicados que explicamos en el capítulo 3 (la difere-rencia es que los indicadores de posición del capítulo 3 se referían a valores de dominio, mien-tras que las variables de alcance en el cálculo de tupias se refieren a las tupias).

En lo que resta del capítulo, daremos por hecho que las definiciones de variable de alcance mostradas arriba están vigentes. Observamos que en un lenguaje real tendrían que existir cier-tas reglas con respecto a tales definiciones. En el presente capítulo ignoramos este aspecto,

Referencias a variables libres y ligadas

Toda referencia a una variable de alcance está ya sea **libre** o **ligada** (en un cierto contexto; e particular, dentro de alguna WFF). Explicaremos primero esta noción en términos puramente sintácticos y después continuaremos con la explicación de por qué es importante. Sea *R* una variable de alcance. Entonces:

- Las referencias a *R* en la WFF "NOT *p"* están libres o ligadas dentro de esa WFF, diendo de si están libres o ligadas en p. Las referencias a *R* en las WFFs "(p AND q)" y "(p
  OR *q)"* están libres o ligadas en esas WFFs, dependiendo de si están libres o ligadas *q,* según sea el caso.
- Las referencias a *R* que están libres en la WFF *"p"* están ligadas en las WFFs "EXISTS R
  (p)" y "FORALL *R* (p)". Otras referencias a variables de alcance en *"p"* están **libres** o das en las WFFs "EXISTS *R* (p)" y "FORALL *R* (p)" dependiendo de si están libres o en *"p".*

Para completar, necesitamos agregar lo siguiente:

- La única referencia a *R* en el <*nombre de variable de alcance*> *"R"* está libre dentn <*nombre de variable de alcance*>.
- La única referencia a *R* en la <*referencia a atributo de alcance*> *"R.A"* está libre dentro de
  esa <*referencia a atributo de alcance*>.

- Si una referencia a *R* está libre en alguna expresión *exp,* esa referencia también estará libre en cualquier expresión *exp'* que contenga inmediatamente a *exp* como una subexpresión, a menos que *exp'* introduzca un cuantificador que haga que la expresión esté ligada.

Por lo tanto, aquí tenemos algunos ejemplos de WFFs que contienen referencias a variables de alcance:

- *Comparaciones simples:*

```
VX.V# = V#  (  'V1'  )

VX.V# = VPX.V# VPX.V#

<> PX.P#
```

En estos ejemplos, todas las referencias a VX, PX y VPX están libres.

- *Combinaciones lógicas de comparaciones simples:*

```
PX.PESO < PESO (  15.5 )  0R PX.CIUDAD =  'Roma'

NOT ( VX.CIUDAD =  'Londres'  )

VX.V# = VPX.V# AND VPX.P# <> PX.P#

PX.COLOR = COLOR  (  'Rojo'  )  OR PX.CIUDAD =  'Londres'
```

De nuevo, aquí están libres todas las referencias a VX, PX y VPX.

- *WFFs cuantificadas:*

```
EXISTS VPX  ( VPX.V# = VX.V# AND VPX.P  =  P#  (  ' P2'  )  )

FORALL PX  ( PX.COLOR = COLOR  (  'Rojo'  )  )
```

En estos dos ejemplos, las referencias a VPX y PX están ligadas y la referencia a VX está libre. Vea la subsección siguiente.

Cuantificadores

Existen dos cuantificadores, EXISTS y FORALL; EXISTS es el cuantificador **existencia!,** FORALL es el cuantificador **universal.** Básicamente, si p es una WFF en la que *R* está libre, entonces tanto

```
EXISTS R ( p )
```

como

```
FORALL R  ( p )
```

son WFFs válidas, y *R* está ligada en ambas. La primera significa: **Existe por lo menos un valor** de *R* que hace que/? dé como resultado *verdadero.* La segunda significa: **Para todos los valores** de *R, p* da como resultado *verdadero.* Por ejemplo, suponga que la variable *R* abarca el conjunto "Miembros del Senado de los Estados Unidos en 1999", y suponga que p es la WFF *"R* es mujer" (por supuesto, aquí no estamos tratando de usar nuestra sintaxis formal). Entonces, "EXISTS *R (p)" y* "FORALL *R (p)"* son WFFs válidas y dan como resultado *verdadero y falso,* respectivamente.

Observe nuevamente el ejemplo de EXISTS del final de la subsección anterior:

```
EXISTS VPX ( VPX.V# = VX.V# AND VPX.P# = P# ( 'P2' ) )
```

De lo anterior se desprende que podemos leer esta WFF de la siguiente forma:

> *Existe una tupia VPX, digamos, en el valor actual de la variable de relación VP tai que el valor V# de esa tupia VPX es igual al valor de VX. V# —cualquiera que sea— y el valor P# en esa tupia VPX es P2.*

Aquí, cada referencia a VPX está ligada. La única referencia a VX está libre.

De manera formal, definimos EXISTS como **un OR iterado.** En otras palabras. una relación con las tupias t1, *t2, ..., tm,* (b) *R* es una variable de alcance que abarca r y (c)p(R) es una WFF en la que *R* ocurre como una variable libre, entonces la WFF

```
EXISTS fl ( p ( fl ) )
```

se define como equivalente a la WFF

```
falso OR p ( tí ) OR ... OR p ( tm
```

Observe en particular que esta expresión da como resultado falso si *r* está vacía (es decir, si m es cero).

A manera de ejemplo, suponga que la relación *r* contiene exactamente las siguientes tupias

```
( 1, 2, 3 )
( 1, 2, 4 )(
1, 3, 4 )
```

Para simplificar, suponga que (a) los tres atributos —en orden de izquierda a derecha, tal como se muestran— se llaman *A, B* y *C,* respectivamente y (b) cada atributo es de tipo INTEGER. Entonces, las siguientes WFFs tienen los valores que se indican:

```
EXISTS fl ( fl.C > 1 )              : verdadero
EXISTS fl ( R.B > 3 )               : falso
EXISTS fl ( R.A > 1 OR fl.C = 4 )   : verdadero
```

Continuemos ahora con FORALL. Aquí repetimos el ejemplo FORALL del final de la sub-sección anterior:

```
FORALL PX ( PX.COLOR = COLOR ( 'Rojo' ) )
```

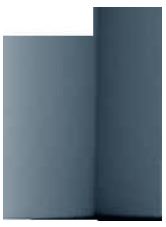
Podemos leer esta WFF como sigue:

> Para todas las tupias PX (por decir algo) en el valor actual de la variable de relación P. el valor de COLOR en esa tupia PX es Rojo.

Aquí, las dos referencias a PX están ligadas.

Así como definimos EXISTS como un OR iterado, del mismo modo definimos FORALL como **un AND iterado.** En otras palabras, si *r, R y p(R)* son como en nuestra explicación ante rior de EXISTS, entonces la WFF

```
FORALL fl ( p ( fl ) )
```



se define como equivalente a la WFF

```
verdadero AND p ( t1 ) AND ... AND p ( tm )
```

En particular, observe que esta expresión da como resultado *verdadero* si r está vacía (es decir, si *m* es cero).

A manera de ejemplo, sea la relación *r* que contiene las mismas tupias que antes. Entonces las WFFs siguientes tienen los valores que se indican:

```
FORALL R ( R.A > 1 )                : falso
FORALL R ( R.B > 1 )                : verdadero
FORALL R ( R.A = 1 AND R.C > 2 )  : verdadero
```

Nota: FORALL es incluido por mera comodidad (no es esencial). Para ser específicos, la identidad

```
FORALL R ( p ) = NOT EXISTS R ( NOT p )
```

(a grandes rasgos, "todas las *R* satisfacen a *p"* es lo mismo que "ninguna *R* no satisface a p") muestra que toda WFF que involucre a FORALL siempre puede ser sustituida por una WFF equivalente que en su lugar involucre a EXISTS. Por ejemplo, la declaración (verdadera)

Para todos los enteros *x,* existe un entero *y* tal que *y > x*

(es decir, todo entero tiene a un entero mayor) es equivalente a la declaración No

existe un entero *x* tal que no exista un entero *y* tal que y > *x*

(es decir, no hay entero alguno que sea el más grande). Pero por lo regular es más fácil pensar en términos de FORALL que en términos de EXISTS o de una doble negación. En otras palabras, en la práctica es deseable soportar ambos cuantificadores.

Revisión de las referencias a variables libres y ligadas

Suponga que *x* abarca el conjunto de todos los enteros, y considere la WFF

```
EXISTS x ( x > 3 )
```

Observe que aquí *x* es una especie de *dummy* que sirve sólo para enlazar la expresión lógica dentro del paréntesis con el cuantificador fuera de él. La WFF establece simplemente que existe algún entero *(x,* por decir algo) que es mayor que tres. *Observe por lo tanto, que el significado de esta WFF permanecería totalmente sin cambio si todas las referencias a x fueran reemplazadas por referencias a alguna otra variable y.* En otras palabras, la WFF

```
EXISTS y ( y > 3 )
```

es semánticamente idéntica a la anterior.

Considere ahora la WFF

```
EXISTS x ( x > 3 ) AND x < 0
```

Aquí tenemos otro ejemplo ("Obtener los nombres de aquellos proveedores que suministran la parte P2"; vea la explicación de EXISTS en la subsección anterior sobre cuantificadores):

```
VX. PROVEEDOR WHERE EXISTS VPX ( VPX.V# = VX.V# AND
                                 VPX.P# = P# ( 'P2' ) )
```

Todas las referencias a VX están libres. Puesto que en la prototupla no hay referencias a la misma variable de alcance, todas las referencias a VPX (en la cláusula WHERE) están ligadas, como debe ser.

De manera intuitiva, una determinada <*operación relacional*> da como resultado una relación que contiene todos los valores posibles de la <*prototupla*> para la cual la <*expresión lógica*> especificada en la cláusula WHERE da como resultado *verdadero* (y omitir la cláusula WHERE equivale a especificar "WHERE *verdadero"*). Para ser más específicos:

- Antes que nada, una prototupla es una posible lista de elementos separados con comas y entre paréntesis en la cual cada elemento es una referencia a un atributo de alcance (y tal vez incluya una cláusula AS para introducir un nuevo nombre de atributo) o bien, un sim ple nombre de variable de alcance.* Sin embargo:

  a. En este contexto, un nombre de variable de alcance es básicamente una forma abrevia da de una lista de referencias a atributos de alcance separadas con comas, una referen cia para cada atributo de la relación que abarca la variable de alcance;

  b. Una referencia a atributo de alcance sin una cláusula AS es básicamente una forma abre viada de otra que incluye dicha cláusula, en la que el nuevo nombre de atributo es el mismo que el anterior.

  Por lo tanto, sin perder la generalidad, podemos considerar a una prototupla como una lista de referencias a atributos de alcance separadas con comas, de la forma *Ri.Aj* AS *Bj*. Observe que no todas las *Ri* y las *Aj* necesitan ser distintas, pero las *Bj* deben serlo.

- Sean *Rl, R2, ..., Rm* las distintas variables de alcance mencionadas en la prototupla. Sean *r1, r2, ..., rm,* respectivamente, las relaciones que abarcan estas variables de alcance. Sean *rl', r2', ..., rm',* respectivamente, las relaciones correspondientes después de aplicar los cambios de nombre de los atributos especificados en las cláusulas AS. Sea r′ el producto cartesiano de *rl', r2', ..., rm'*.

- Sea r la restricción de *r'* que satisface la WFF de la cláusula WHERE. *Nota:* Para efectos de la explicación, damos por hecho que los cambios de nombre del paso anterior se aplican también a los atributos mencionados en la cláusula WHERE, ya que de otro modo podría no tener sentido la WFF de esa cláusula WHERE. Sin embargo, nuestra sintaxis concreta de hecho no depende de esta suposición sino que depende de la calificación de puntos para manejar cualquier ambigüedad que se presentara, como veremos en la siguiente sección.

- El valor general de la <*operación relacional*> se define como la proyección de r sobre todas las *Bj*.

Consulte algunos ejemplos en la sección siguiente.

---

*A1 menos en forma general (por el momento limitaremos nuestra atención únicamente a estas dos posibilidades).

7.3 EJEMPLOS

Presentamos algunos ejemplos del uso del cálculo en la formulación de consultas. Como ejercicio, también podría intentar dar soluciones algebraicas, para fines de "comparación y contraste".

7.3.1 Obtener los números de proveedor y el status de los proveedores de París con **status > 20.**

```
 ( VX.V#, VX.STATUS )
WHERE VX. CIUDAD = 'París' AND VX. STATUS > 20
```

7.3.2 Obtener todos los pares de números de proveedor tales que los dos proveedores estén **coubicados (es decir, ubicados en la misma ciudad).**

```
 ( VX.V# AS VA, VY.V# AS VB )
               WHERE VX.CIUDAD = VY.CIUDAD AND VX.V# < VY.V#
```

Observe que las cláusulas AS de la prototupla dan nombres a los atributos del *resultado;* por lo tanto dichos nombres no están disponibles para su uso en la cláusula WHERE, por lo cual la se-gunda comparación en esa cláusula WHERE es "VX.V# < VY.V#" y no "VA < VB".

7.3.3 Obtener toda la información de aquellos proveedores que suministran la parte P2.

```
VX
WHERE EXISTS VPX ( VPX.V# = VX.V# AND VPX.P# = P# ( ' P2' ) )
```

Aquí observe el uso de un nombre de variable de alcance en la prototupla. El ejemplo es una forma abreviada de lo siguiente:

```
 ( VX.V#, VX.PROVEEDOR, VX.STATUS, VX.CIUDAD )
WHERE EXISTS VPX ( VPX.V# - VX.V# AND VPX.P# = P# ( 'P2'
```

7.3.4 Obtener los nombres de aquellos proveedores que suministran por lo menos una **parte roja.**

```
VX.PROVEEDOR
WHERE EXISTS VPX ( VPX.V# = VX.V# AND
                   EXISTS PX ( PX.P# = VPX.P# AND
                               PX.COLOR = COLOR ( 'Rojo' ) ) )
```

O de manera equivalente (pero en la **forma normal prenexa,** en la que todos los calificadores aparecen al frente de la WFF):

```
VX. PROVEEDOR
WHERE EXISTS VPX ( EXISTS PX ( VX.V# = VPX.V# AND
                               VPX.P# = PX.P# AND PX.COLOR =
                               COLOR ( 'Rojo' ) ) )
```

La forma normal prenexa no es, de manera inherente, ni más ni menos correcta que cualquier otra forma, pero con un poco de práctica tiende a ser en muchos casos la formulación más natural. Además, presenta la posibilidad de reducir el número de paréntesis, como sigue. La WFF

cuantif1 vble1 (cuantif2 vble2 (wff))

(donde *cuantifl* y *cuantif2* pueden ser EXISTS o bien, FORALL) puede ser abreviada de manera opcional y sin ambigüedad a sólo

cuantifl vble1 cuantif2 vble2 (w*ff)* De esta manera, podemos simplificar la

expresión de cálculo que mostramos arriba a sólo

```
VX.PROVEEDOR
WHERE EXISTS VPX EXISTS PX ( VX.V# = VPX.V# AND
                             VPX.P# = PX.P# AND
                             PX.COLOR • COLOR ( 'Rojo' ) )
```

Sin embargo, por razones de claridad, en esta sección seguiremos mostrando todos los paréntesis de manera explícita.

7.3.5 Obtener los nombres de aquellos proveedores que suministran por lo menos una de las partes que suministra el proveedor V2.

```
VX.PROVEEDOR
WHERE EXISTS VPX ( EXISTS VPY ( VX.V# • VPX.V# AND
                                VPX.P# = VPY.P# AND
                                VPY.V# = V# ( 'V2' ) ) )
```

7.3.6 Obtener los nombres de los proveedores que suministran todas las partes.

```
VX. PROVEEDOR WHERE FORALL PX ( EXISTS VPX ( VPX.V# = VX.V# AND
                                            VPX.P# = PX.P# ) )
```

O de manera equivalente (pero sin usar FORALL):

```
VX. PROVEEDOR WHERE NOT EXISTS PX ( NOT EXISTS VPX
                                        ( VPX.V# • VX.V# AND
                                          VPX.P# ■ PX.P# ) )
```

7.3.7 Obtener los nombres de aquellos proveedores que no suministran la parte P2.

```
VX.PROVEEDOR WHERE NOT EXISTS VPX
                    ( VPX.V# = VX.V# AND VPX.P# = P# ( 'P2' ) )
```

Observe con qué facilidad se deriva esta solución de la correspondiente al ejemplo 7.3.3 anterior.

7.3.8 Obtener los números de los proveedores que suministran por lo menos todas las partes que suministra el proveedor V2.

```
VX.V# WHERE FORALL VPX ( VPX.V# <> V# ( ' V2' ) OR
                         EXISTS VPY ( VPY.V# - VX.V# AND
                                      VPY.P# = VPX.P# ) )
```

O lo que es lo mismo: "Obtener los números de los proveedores VX tales que para todos los envíos VPX, ya sea que ese envío no sea del proveedor V2, o si lo es, exista entonces un envío VPY de la parte VPX del proveedor VX."

Presentamos otra forma sintáctica abreviada para ayudar en consultas complejas como ésta; es decir, una forma sintáctica explícita del operador de **implicación lógica.** Si *p* y *q* son WFFs, entonces la expresión de implicación lógica

```
IF p THEN q END IF
```

también es una WFF, con una semántica idéntica a la de la WFF

```
( NOT p ) OR q
```

Por lo tanto, el ejemplo puede ser reformulado como sigue:

```
VX.V# WHERE FORALL VPX ( IF VPX.V# = V# ( 'V2'  ) THEN
                          EXISTS VPY ( VPY.V# ■ VX.V# AND
                                       VPY.P# = VPX.P# )
                          END IF )
```

O lo que es lo mismo: "Obtener todos los números de aquellos proveedores VX tales que, para todos los envíos VPX, si ese envío VPX es del proveedor V2, exista entonces un envío VPY la parte VPX del proveedor VX."

7.3.9 Obtener los números de las partes que pesen más de 16 libras o **que sean suministradas por el proveedor V2, o ambas cosas.**

```
RANGEVAR PU RANGES OVER
       ( PX.P# WHERE PX.PESO > PESO ( 16.0 ) ),
       ( VPX.P# WHERE VPX.V# = V# ( 'V2' ) ) ;
PU.P#
```

Aquí, el álgebra relacional incluiría una unión explícita.

Por interés, mostramos una formulación alternativa para esta consulta. Sin embargo, esta segunda formulación depende del hecho de que todo número de parte en la varrel VP aparea también en la varrel P, lo cual no hace la formulación al "estilo unión".

```
PX.P# WHERE PX.PESO > PESO ( 16.0 )
      OR   EXISTS VPX ( VPX.P# " PX.P# AND
                        VPX.V# = V# ( 'V2' ) )
```

7.4 EL CALCULO FRENTE AL ALGEBRA

En la introducción de este capítulo afirmamos que el álgebra y el cálculo son fundamentalmente equivalentes. Ahora examinaremos con más detalle esa afirmación. Primero. Codd mostréenla referencia [6.1] que el álgebra es al menos tan poderosa como el cálculo. Esto lo hizo dando un algoritmo —"el algoritmo de reducción de Codd"— mediante el cual una expresión arbitral del cálculo podía ser reducida a una expresión del álgebra semánticamente equivalente. No presentamos aquí el algoritmo de Codd en detalle, sino que nos conformamos con un ejemplo razonablemente complejo que ilustra a grandes rasgos cómo funciona el algoritmo.*

---

*En realidad, el algoritmo presentado en la referencia [6.1] tenía un ligero defecto [7.2], Además la ver-sión del cálculo definida en ese artículo no incluía una contraparte completa para el operador de unión, así que de hecho el cálculo de Codd era estrictamente menos poderoso que el álgebra de Codd. Sin e como varios autores han demostrado, es cierta la afirmación de que el álgebra y el cálculo —mejorado éste para incluir una contraparte completa para la unión— son equivalentes; vea por ejemplo, Klug [6.12],

Como una base para nuestro ejemplo no usamos la conocida base de datos de proveedores y partes, sino la versión ampliada de proveedores, partes y proyectos que aparece en los ejercicios del capítulo 4 y en otras partes del libro. Por comodidad, en la figura 7.1 mostramos un conjunto de valores de ejemplo para esa base de datos (repetidos de la figura 4.5 del capítulo 4).

Ahora considere la consulta "Obtener los nombres y ciudades de los proveedores que suministran por lo menos a un proyecto en Atenas con por lo menos 50 de cada una de las partes". Una expresión de cálculo para esta consulta es:

```
( VX. PROVEEDOR, VX.CIUDAD ) WHERE EXISTS YX FORALL PX EXISTS VPYX
                                        ( YX.CIUDAD ■ 'Atenas' AND
                                          YX.Y# = VPYX.Y# AND
                                          PX.P# = VPYX.P# AND
                                          VX.V# " VPYX.V# AND
                                          VPYX.CANT > CANT ( 50 ) )
```

en donde VX, PX, YX y VPYX son variables de alcance que abarcan a V, P, Y y VPY, respectivamente. Ahora mostramos cómo podemos evaluar esta expresión para producir el resultado deseado.

Paso 1: Para cada variable de alcance, recuperar el alcance (es decir, el conjunto de valores posibles para esa variable), restringido de ser posible. Con "de ser posible restringido", queremos decir que podría existir una condición de restricción incrustada dentro de la cláusula

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

p#	PARTE	COLOR	PESO	CIUDAD
P1	Tuerca	Rojo	12.0	Londres
P2	Perno	Verde	17.0	París
P3	Tornillo	Azul	17.0	Roma
P4	Tornillo	Rojo	14.0	Londres
P5	Leva	Azul	12.0	París
P6	Engrane	Rojo	19.0	Londres

Y#	PROYECTO	CIUDAD
Y1	Clasificador	París
Y2	Monitor	Roma
Y3	OCR	Atenas
Y4	Consola	Atenas
Y5	RAID	Londres
Y6	EDS	Oslo
Y7	Cinta	Londres

VPY

V#	P#	Y#	CANT
V1	P1	Y1	200
V1	P1	Y4	700
V2	P3	Y1	400
V2	P3	Y2	200
V2	P3	Y3	200
V2	P3	Y4	500
V2	P3	Y5	600
V2	P3	Y6	400
V2	P3	Y7	800
V2	P5	Y2	100
V3	P3	Y1	200
V3	P4	Y2	500
V4	P6	Y3	300
V4	P6	Y7	300
V5	P2	Y2	200
V5	P2	Y4	100
V5	P5	Y5	500
V5	P5	Y7	100
V5	P6	Y2	200
V5	P1	Y4	100
V5	P3	Y4	200
V5	P4	Y4	800
V5	P5	Y4	400
V5	P6	Y4	500

Figura 7.1 La base de datos proveedores, partes y proyectos (valores de ejemplo).

WHERE que pueda usarse de inmediato para eliminar ciertas tupias de toda consideración teror. En el caso que nos ocupa, los conjuntos de tupias recuperadas son los siguientes:

VX	:	Todas las tupias de V	5 tupias
PX	:	Todas las tupias de P	6 tupias
YX	:	Las tupias de Y donde CIUDAD = 'Atenas'	2 tupias
VPYX	:	Las tupias de VPY donde CANT >= CANT (50)	24 tupias

Paso 2: Construir el producto cartesiano de los rangos recuperados en el paso 1. pan producir:

V	PROVE	STA	CIU	P#	PARTE	COLOR	PESO	CIU	Y#	PROY	CIU	V#	p#	V»	CA
V1	Smith	20	Lon	P1	Tuerc	Rojo	12.0	Lon	Y3	OCR	Ate	V1	P1	Y1	200
V1	Smith	20	Lon	P1	Tuerc	Rojo	12.0	Lon	Y3	OCR	Ate	V1	P1	Y4	700

(etcétera). El producto completo contiene 5 * 6 * 2 * 24= 1440 tupias. *Nota:* Por razones de espacio, hemos realizado algunas abreviaturas obvias. Además, no nos molestamos en renombrar los atributos (como en realidad debimos hacerlo, para evitar la ambigüedad); en vez de ello con-fiamos en la posición ordinal para mostrar, por ejemplo, qué "V#" proviene de V y cuál de VPY Adoptamos este truco poco ortodoxo para abreviar simplemente la exposición.

Paso 3: Restringir el producto cartesiano construido en el paso 2 de acuerdo con la "por-ción de junta" de la cláusula WHERE. En el ejemplo, dicha porción es

```
YX.Y# • VPYX.Y# AND PX.P# = VPYX.P# AND VX.V# = VPYX.V#
```

Por lo tanto, eliminamos del producto las tupias para las cuales el valor V# de proveedores es igual al valor V# de envíos, o el valor P# de partes no es igual al valor P# de envíos, o el valor Y# de proyectos no es igual al valor Y# de envíos, para producir un subconjunto del producto cartesiano (tal como se ve) de sólo diez tupias:

V	PROVE	STA	CIU	P#	PARTE	COLOR	PESO	CIU	Y#	PROY	CIU	V	P#	Y#	CA
V1	Smith	20	Lon	P1	Tuerc	Rojo	12.0	Lon	Y4	Cons	Ate	V1	P1	Y4	700
V2	Jones	10	Par	P3	Torni	Azul	17.0	Rom	Y3	OCR	Ate	V2	P3	Y2	
V2	Jones	10	Par	P3	Torni	Azul	17.0	Rom	Y4	Cons	Ate	V2	P3	Y?	
V4	Clark	20	Lon	P6	Engra	Rojo	19.0	Lon	Y3	OCR	Ate	V4	P6	Y»	
V5	Adams	30	Ate	P2	Perno	Verde	17.0	Par	Y4	Cons	Ate	V5	P2	Y1	
V5	Adams	30	Ate	P1	Tuerc	Rojo	12.0	Lon	Y4	Cons	Ate	V5	P1	Y1	
V5	Adams	30	Ate	P3	Torni	Azul	17.0	Rom	Y4	Cons	Ate	V5	P3	Y2	
V5	Adams	30	Ate	P4	Torni	Rojo	14.0	Lon	Y4	Cons	Ate	V5	P4	Y8	
V5	Adams	30	Ate	P5	Leva	Azul	12.0	Par	Y4	Cons	Ate	V5	P5	Y*	
V5	Adams	30	Ate	P6	Engra	Rojo	19.0	Lon	Y4	Cons	Ate	V5	P6	Y4	

(Por supuesto, esta relación es la equijunta pertinente.)

Paso 4: Aplicar los cuantificadores de derecha a izquierda como sigue.

- Para el cuantificador "EXISTS *RX"* (donde *RX* es una variable de alcance que abarca ci(relación r), *proyectar* el resultado intermedio actual para eliminar todos los atributos de la relación r.

- Para el cuantificador "FORALL *RX', dividir* el resultado intermedio actual entre la relación de "rango restringido" asociada con *RX* tal como fue recuperada en el paso 1. Esta operación también tendrá el efecto de eliminar todos los atributos de la relación *r. Nota:* Aquí "dividir" significa la operación de división original de Codd (vea el comentario de la referencia [6.3]).

  En el ejemplo, los cuantificadores son:

```
EXISTS YX FORALL PX EXISTS VPYX
```

Por lo tanto:

1. *(EXISTS VPYX)* Proyecta los atributos de VPY hacia afuera de la relación; es decir, VPY.V#, VPY.P#, VPY.Y# y VPY.CANT. *Resultado:*

V #	PROVE	STA	CIU	P#	PARTE	COLOR	PESO	CIU	Y#	PROY	CIU
V1	Smith	20	Lon	P1	Tuerc	Rojo	12.0	Lon	Y4	Cons	Ate
V2	Jones	10	Par	P3	Torni	Azul	17.0	Rom	Y3	OCR	Ate
V2	Jones	10	Par	P3	Torni	Azul	17.0	Rom	Y4	Cons	Ate
V4	Clark	20	Lon	P6	Engra	Rojo	19.0	Lon	Y3	OCR	Ate
V5	Adams	30	Ate	P2	Perno	Verde	17.0	Par	Y4	Cons	Ate
V5	Adams	30	Ate	P1	Tuerc	Rojo	12.0	Lon	Y4	Cons	Ate
V5	Adams	30	Ate	P3	Torni	Azul	17.0	Rom	Y4	Cons	Ate
V5	Adams	30	Ate	P4	Torni	Rojo	14.0	Lon	Y4	Cons	Ate
V5	Adams	30	Ate	P5	Leva	Azul	12.0	Par	Y4	Cons	Ate
V5	Adams	30	Ate	P6	Engra	Rojo	19.0	Lon	Y4	Cons	Ate

2. *(FORALL PX)* Divide entre P. *Resultado:*

V#	PROVEEDOR	STATUS	CIUDAD	Y#	PROYECTO	CIUDAD
V5	Adams	30	Atenas	Y4	Consola	Atenas

   (Ahora disponemos de espacio para mostrar el resultado sin abreviaturas).

3. *(EXISTS YX)* Proyectar los atributos de Y hacia afuera de la relación; es decir, Y.Y#, Y.PROYECTO y Y.CIUDAD. *Resultado:*

V#	PROVEEDOR	STATUS	CIUDAD
V5	Adams	30	Atenas

Paso 5: Proyectar el resultado del paso 4 de acuerdo con las especificaciones de la prototupla. En nuestro ejemplo, la prototupla es:

```
( VX.PROVEEDOR, VX.CIUDAD )
```

Por lo tanto, nuestro resultado final es:

PROVEEDOR	CIUDAD
Adams	Atenas

De todo lo anterior se desprende que la expresión original del cálculo es semánticamente
equivalente a una cierta expresión algebraica anidada (para ser precisos, una proyección de proyección de una división de una proyección de una restricción de un producto de cuatro tricciones).

Esto concluye el ejemplo. Por supuesto, es posible mejorar el algoritmo de muchas (consulte el capítulo 17 —en particular la referencia [17.5]— para algunas ideas de dichas ras) y en nuestra explicación omitimos muchos detalles; sin embargo, el ejemplo debe ser ciente para dar la idea general de cómo funciona la reducción.

Por cierto, ahora podemos explicar una de las razones (no la única) por las que Codd definió precisamente los ocho operadores algebraicos que hizo. Esos ocho operadores proporciona **lenguaje destino** que sirve como vehículo para una posible implementación del cálculo. En palabras, dado un lenguaje como QUEL que está fundamentado en el cálculo, un enfoque posi-ble para implementar ese lenguaje sería tomar la consulta como la envió el usuario —lo cual es básicamente sólo una expresión de cálculo— y aplicarle el algoritmo de reducción | obtener una expresión algebraica equivalente. Por supuesto, esa expresión algebraica consiste en un conjunto de operadores algebraicos, los cuales (por definición) son inherentemente imple-mentables. (El siguiente paso consiste en *optimizar* esa expresión algebraica. Vea el c

Otro punto importante es que los ocho operadores algebraicos de Codd también (*parámetro* para medir el poder expresivo de cualquier lenguaje de base de datos < Mencionamos este aspecto brevemente en el capítulo 6, al final de la sección 6.6; aha minaremos un poco más a fondo.

Primero, se dice que un lenguaje es **relacionalmente completo** si es por lo menos t roso como el cálculo; es decir, si cualquier relación definible por alguna expresión del c también es definible por alguna expresión del lenguaje en cuestión [6.1]. En el capítulo 6 dijimos que "relacionalmente completo" significaba tan poderoso como el *álgebra,* no como el c (aunque viene a ser lo mismo, como veremos en un momento). Observe que por la existencia del algoritmo de reducción de Codd se desprende inmediatamente que el álgebra es mente completa.

La compleción relacional puede ser considerada como una medida básica del poder expre-sivo de los lenguajes de base de datos en general. En particular, puesto que el cálculo y bra son relacionalmente completos, ambos proporcionan una base para diseñar 1 ofrezcan este poder de expresividad *sin tener que recurrir al uso de ciclos.* (Una consideración particularmente importante en el caso de un lenguaje destinado a usuarios finales, auque tam-bien es importante para los programadores de aplicaciones.)

Entonces, puesto que el álgebra es relacionalmente completa, entendemos que, para que cualquier lenguaje *L* también es completo, basta con mostrar (a) que *L* incluye equiv, los ocho operadores algebraicos (de hecho, es suficiente con mostrar que incluye equiv de los cinco operadores algebraicos *primitivos)* y (b) que los operandos de cualquier c en *L* pueden ser expresiones arbitrarias de *L.* SQL es un ejemplo de un lenguaje que puede mos-trar ser razonablemente completo de esta forma —vea el ejercicio 7.9— y QUEL es c hecho, en la práctica generalmente es más fácil mostrar que un lenguaje dado tiene equiv; de los operadores algebraicos, a mostrar que tiene equivalentes de las expresiones del c Es por esto que definimos generalmente la compleción relacional en términos algebraicos en vez de términos de cálculo.

Por cierto, hay que entender que la compleción relacional no implica necesariamente quier otra clase de compleción. Por ejemplo, es necesario que un lenguaje también ofrezca "com-

pleción computational"; es decir, debe ser capaz de calcular todas las funciones computables. La completion computational fue una de las motivaciones por las que en el capítulo 6 incorporamos los operadores EXTEND y SUMMARIZE al álgebra. En la siguiente sección consideraremos operadores de cálculo equivalentes a ellos.

Para retomar el asunto de la equivalencia entre el álgebra y el cálculo, hemos mostrado por medio de un ejemplo que cualquier expresión de cálculo puede ser reducida a un equivalente algebraico y por lo tanto, que el álgebra es al menos tan poderosa como el cálculo. De manera inversa, es posible mostrar que cualquier expresión algebraica puede ser reducida a un equivalente del cálculo; para una muestra de ello, vea por ejemplo a Ullman [7.13]. Deducimos entonces que ambos son lógicamente equivalentes.

7.5 POSIBILIDADES COMPUTACIONALES

Anteriormente no lo señalamos de manera explícita, pero de hecho el cálculo tal como lo hemos definido, ya incluye equivalentes de los operadores algebraicos EXTEND y SUMMARIZE, debido a que:

- Una posible forma de la prototupla es una <*invocación al selector de tupla*>, y los componentes de esta invocación pueden ser <*expresiones*> arbitrarias.
- Los comparandos de una comparación en una <*expresión lógica*> también pueden ser <*expresiones*> arbitrarias.
- El primero o único argumento para una <*invocación a operador de totales*> es una <*operación relacionate.*

Nota: Acudimos aquí a la definición completa de **Tutorial D** tal como está dada en la referencia [3.3].

No vale la pena entrar aquí en todos los detalles sintácticos y semánticos aplicables. Simplemente nos conformamos con dar algunos ejemplos (los cuales están ligeramente simplificados en ciertos aspectos).

7.5.1 Obtener el número de parte y el peso en gramos de cada parte con peso > 10000 gramos.

```
( PX.P#, PX.PESO * 454 AS PSGR )
                WHERE PX.PESO *454 > PESO ( 10000.0 )
```

Observe que la especificación "AS PSGR" en la prototupla asigna un nombre al atributo aplicable del *resultado.* Por lo tanto, el nombre no está disponible para su uso en la cláusula WHERE, razón por la cual aparece dos veces la expresión "PX.PESO * 454".

7.5.2 Obtener todos los proveedores y etiquetar a cada uno con el valor literal "Proveedor".

```
( VX,  'Proveedor' AS TAG )
```

7.5.3 Para cada envío, obtener todos los detalles incluyendo el peso total del mismo.

```
( VPX, PX.PESO * VPX.CANT ) AS PSENV WHERE PX.P# = VPX.P#
```

7.5.4 Para cada parte, obtener el número de parte y la cantidad total del envío.

```
( PX.P#, SUM ( VPX WHERE VPX.P# = PX.P#, CANT ) AS CANTOT )
```

7.5.5 Obtener la cantidad total de envíos

```
SUM ( VPX, CANT ) AS GRANTOTAL
```

7.5.6 Para cada proveedor, obtener el número de proveedor y el número total de partes **suministradas.**

```
( VX.V#, COUNT ( VPX WHERE VPX.V# = VX.V# ) AS #_DE_PARTES )
```

7.5.7 Obtener las ciudades de las partes que almacenan más de cinco partes rojas.

```
RANGEVAR PY (RANGES OVER P ;

PX.CIUDAD
WHERE COUNT ( PY WHERE PY.CIUDAD = PX.CIUDAD
                AND  PY.COLOR = COLOR ( 'Rojo1 ) ) > 5
```

7.6 CALCULO DE DOMINIOS

Como mencionamos en la sección 7.1, el cálculo de dominios difiere del cálculo de tuplas en que sus variables de alcance abarcan *dominios* en lugar de relaciones. En este libro explicamos el cálculo de dominios de manera más bien breve. Desde un punto de vista práctico, la diferen-cia de sintaxis que resulta más obvia inmediatamente es que el cálculo de dominios maneja una forma adicional de <*expresión lógica*> a la cual nos referiremos como **condición** de perte-**nencia.** Una condición de pertenencia toma la forma

```
R ( par, par, ... )
```

donde *R* es un nombre de varrel y cada *par es* de la forma *A:v,* donde *A* es un atributo de *R y v* puede ser el nombre de una variable de alcance del cálculo de dominios, o bien una invocación a selector (por lo regular una literal). La condición da como resultado *verdadero* si, y **sólo:** existe una tupia en cualquier relación que sea el valor actual de *R* y que tenga los valores esf cificados para los atributos señalados. Por ejemplo, la expresión

```
VP  ( V#:V#('V1'),  P#:P#('P1')  )
```

es una condición de pertenencia que da como resultado *verdadero* si, y sólo si, existe actualmente una tupia de envío con el valor V1 en V# y el valor P1 en P#. En forma similar, la condición de pertenencia

```
VP ( V#:VX,  P#:PX )
```

da como resultado *verdadero* si, y sólo si, existe actualmente una tupia de envío con un valor de V# igual al valor actual de la variable de alcance VX (cualquiera que éste pueda ser) y un valor de P# igual al valor actual de la variable de alcance PX (cualquiera que éste pueda ser).

Para el resto de esta sección supondremos la existencia de variables de alcance para cálculo de dominios, como sigue:

Dominio:	*Variables de alcance:*
V# P#	VX, VY, ...
NOMBRE	PX, PY, ... NOMX,
COLOR	NOMY, ... COLORX,
PESO	COLOP.Y, . . . PESOX,
CANT	PESOY, ... CANTX,
CHAR	CANTY, ... CIUDADX,
INTEGER	CIUDADY, .. STATUSX,
	STATUSY, ..

Entonces, aquí tenemos algunos ejemplos de expresiones del cálculo de dominios:

```
VX

VX WHERE V ( V#:VX )

VX WHERE V ( V#:VX, CIUDAD:'Londres' )

( VX, CIUDADX ) WHERE V ( V#:VX, CIUDAD:CIUDADX )
                AND    VP ( V#:VX, P#:P#('P2') )

( VX, PX ) WHERE V ( V#:VX, CIUDAD:CIUDADX )
           AND P ( P#:PX, CIUDAD:CIUDADY )
           AND   CIUDADX <> CIUDADY
```

Hablando a grandes rasgos, la primera de estas expresiones denota el conjunto de todos los números de proveedor; la segunda denota el conjunto de todos los números de proveedor en la varrel V; la tercera denota el subconjunto de esos números de proveedor para los que la ciudad es Londres. La siguiente es una representación del cálculo de dominios de la consulta "Obtener los números de proveedor y las ciudades de los proveedores que suministran la parte P2" (observe que la versión del cálculo de tupias de esta consulta requirió de un cuantificador existencial). La última es una representación del cálculo de dominios de la consulta "Obtener los pares número de proveedor/número de parte tales que el proveedor y la parte no estén coubicados".

Damos las versiones del cálculo de dominios de algunos de los ejemplos de la sección 7.3 (algunos ligeramente modificados).

7.6.1 Obtener los números de los proveedores en París con status > 20.

```
VX WHERE EXISTS STATUSX
       ( STATUSX > 20 AND V (
         V#:VX, STATUS:STATUSX,      CIUDAD: 'París' ) )
```

El primer ejemplo es en cierta forma más burdo que su contraparte en el cálculo de tupias (en particular, observe que aún se requiere de cuantificadores). Por otra parte, también hay casos en los que sucede lo contrario; en especial, vea algunos de los ejemplos más complejos que aparecen más adelante en esta sección.

7.6.2 Obtener todos los pares de números de proveedor tales que los dos proveedores estén coubicados.

```
( VX AS VA, VY AS VB ) WHERE EXISTS CIUDADZ
                             ( V ( V#:VX, CIUDAD:CIUDADZ ) AND
                               V ( V#:VY, CIUDAD:CIUDADZ ) AND
                               VX < VY )
```

7.6.3 Obtener los nombres de los proveedores que suministran por lo menos una parte **roja.**

```
NOMX WHERE EXISTS VX EXISTS PX
     ( V ( V#:VX, PROVEEDOR:NOMX ) AND VP (
       V#:VX, P#:PX ) AND P ( P#:PX,
       COLOR:COLOR('Rojo') ) )
```

7.6.4 Obtener los nombres de los proveedores que suministran por lo menos una parte **las que suministra el proveedor V2.**

```
NOMX WHERE EXISTS VX EXISTS PX
     ( V ( V#:VX, PROVEEDOR:NOMX ) AND
       VP ( V#:VX, P#:PX ) AND VP (
       V#:V#('V2'), P#:PX ) )
```

7.6.5 Obtener los nombres de los proveedores que suministran todas las partes.

```
NOMX WHERE EXISTS VX ( V ( V#:VX, PROVEEDOR:NOMX ) AND
      FORALL PX ( IF P ( P#:PX )
                     THEN VP ( V#:VX, P#:PX )
                     END IF ) )
```

7.6.6 Obtener los nombres de los proveedores que no suministran la parte P2.

```
NOMX WHERE EXISTS VX ( V ( V#:VX, PROVEEDOR:NOMX )
                       AND NOT VP ( V#:VX, P#:P#('P2') ) )
```

7.6.7 Obtener los números de proveedor de los proveedores que suministran por lo me **todas las partes que suministra el proveedor V2.**

```
VX WHERE FORALL PX ( IF VP ( V#:V#('V2'), P#:PX ) THEN
                     VP ( V#:VX, P#:PX ) END IF )
```

7.6.8 Obtener los números de parte de las partes que pesan más de 16 libras o **que** son **ministradas por el proveedor V2, o ambas cosas.**

```
PX WHERE EXISTS PESOX
     ( P ( P#:PX, PESO:PESOX ) AND
     PESOX > PESO ( 16.0 ) ) OR VP (
     V#: V# ( ' V2 ' ), P#: PX )
```

El cálculo de dominios, al igual que el de tupias es formalmente equivalente al álgebra re-lacional (es decir, es relacionalmente completo). Para una demostración, vea por ejemplo a **Ullman [7.13].**

7.7 PROPIEDADES DE SQL

En la sección 7.4 mencionamos que un lenguaje relacional dado podría estar basado ya» el álgebra relacional o en el cálculo relacional. Entonces, ¿en cuál se basa SQL? La respuesta por desgracia, es parcialmente en ambos y parcialmente en *ninguno...* Cuando se diseño por primera vez, SQL tenía la intención específica de ser diferente tanto del álgebra como del cálcu-lo [4.8]; de hecho, dicha meta fue la motivación principal para la introducción de la **construc** ción "IN *<subconsulta>"* (vea el ejemplo 7.7.10, más adelante en esta sección). Sin emba con el paso del tiempo resultó que después de todo eran necesarias ciertas características tanto

del álgebra como del cálculo, y el lenguaje creció para darles lugar.* Por lo tanto, la situación actual es que algunos aspectos de SQL están estructurados "al estilo del álgebra", algunos "al estilo del cálculo" y otros más a ninguno de ellos. Esta situación explica por qué mencionamos en el capítulo 6 que diferiríamos al capítulo actual la explicación de las propiedades de manipulación de datos de SQL. (Dejamos como ejercicio determinar qué partes de SQL se basan en el álgebra, cuáles en el cálculo y cuáles en ninguno de ellos.)

Una consulta de SQL se formula como una **expresión de tabla,** de una complejidad potencialmente considerable. No abordaremos aquí toda esa complejidad; más bien, sólo presentamos un conjunto de ejemplos, con la idea que dichos ejemplos resalten los puntos más importantes. Los ejemplos están basados en las definiciones de las tablas de SQL para proveedores y partes que mostramos en el capítulo 4 (figura 4.1). En particular, recuerde que en la versión de SQL para esa base de datos no hay tipos de datos definidos por el usuario; en su lugar, todas las columnas están definidas en términos de alguno de los tipos integrados de SQL. *Nota:* En el apéndice A aparece un tratamiento más formal de las expresiones de SQL en general, y de las expresiones de tablas de SQL en particular.

7.7.1 Obtener el color y la ciudad de todas las partes "no de París" con un peso mayor a 10 libras.

```
SELECT PX.COLOR, PX.CIUDAD
FROM  P AS PX WHERE
PX.CIUDAD <> 'París' AND
PX.PESO > 10.0 ;
```

Puntos a destacar:

1. En este ejemplo, observe el uso del operador de comparación "o" (desigual). Los opera dores de comparación escalares usuales en SQL son: =, <>, <, >, <=, >=.
2. Observe también la especificación "P AS PX" en la cláusula FROM. Esta especificación constituye en efecto la definición de una variable de alcance (al estilo del cálculo de tupias) denominada PX, la cual abarca el valor actual de la tabla P. El alcance de esta definición es, a grandes rasgos, la expresión de tabla en la que aparece. *Nota:* A PX, SQL lo llama un *nombre de correlación.*
3. SQL también soporta la idea de variables de alcance *implícitas,* de acuerdo con lo cual la consulta que nos ocupa podría haberse expresado igualmente bien de la siguiente manera:

```
SELECT  P.COLOR, P.CIUDAD
FROM          P
WHERE   P.CIUDAD <> 'París'
AND     P.PESO > 10.0 ;
```

La idea básica consiste en permitir la utilización de un nombre de tabla para denotar una variable de alcance implícita que abarque la tabla en cuestión (por supuesto, con la condición de

---

*Una consecuencia de ese crecimiento es que —como señala la anotación a la referencia [4.18]— ahora se podría quitar del lenguaje toda la construcción "IN *<subconsulta>*", ¡sin perder funcionalidad! Este hecho es irónico, ya que fue esa construcción a la que se refería el término "Estructurado" del nombre original "Lenguaje de Consulta Estructurado"; de hecho, esa construcción fue en primer lugar la justificación original para adoptar SQL en vez del álgebra o el cálculo.

que no resulte una ambigüedad). En el ejemplo, la parte FROM P, puede ser considerada como una forma abreviada de una cláusula FROM que se lee como FROM P AS P. En otras palabras, debemos entender con claridad que aquí la "P" (por ejemplo) en "P.COLOR' de las cláusulas SELECT y WHERE *no* se refiere a la tabla P, sino que se refiere a una *ramble de alcance* denominada P que abarca a la tabla del mismo nombre.

4. Como señalamos en el capítulo 4, a lo largo de este ejemplo podríamos haber usado non bres de columnas sin calificar, escribiendo:

```
SELECT COLOR, CIUDAD
FROM   P
WHERE CIUDAD <> 'París'
AND    PESO > 10.0 ;
```

   La regla general es que los nombres sin calificar son aceptables si no causan ambigüedad. Sin embargo, en nuestros ejemplos incluiremos generalmente todos los calificadores, aun cuando sean técnicamente redundantes. No obstante, por desgracia hay ciertos contextos en los que se requiere explícitamente que los nombres de columnas *no* estén calificados. La cláusula ORDER BY es uno de estos casos. (Vea el ejemplo que sigue.)

5. La cláusula ORDER BY, mencionada en el capítulo 4 en conexión con DECLARE CURSO también puede ser usada en consultas interactivas de SQL. Por ejemplo:

```
SELECT P.COLOR, P.CIUDAD
FROM   P
WHERE P.CIUDAD <> 'París'
AND    P.PESO > 10.0
ORDER BY CIUDAD DESC ;
```

6. Le recordamos la forma abreviada "SELECT *", mencionada también en el capítulo 4.Por ejemplo:

```
SELECT *
FROM   P
WHERE  P.CIUDAD <> 'París'
AND    P.PESO > 10.0 ;
```

   El asterisco en "SELECT *" es una forma abreviada de una lista separada con comas de todos los nombres de columna de las tablas a las que la cláusula FROM hace re en el orden de izquierda a derecha en el que dichas columnas están definidas dentro dea tablas. Subrayamos que la notación de asterisco es conveniente para las consultas interactivas, ya que ahorra captura. Sin embargo, es potencialmente peligrosa en el SQL incrus-tado —es decir, el SQL incrustado en un programa de aplicación—, ya que el significado del "*" podría cambiar (por ejemplo, si se agrega o elimina una columna en una tabla ,me-diante ALTER TABLE).

7. *(¡Mucho más importante que los puntos anteriores!)* Observe que, dados nuestros datos ejemplo usuales, la consulta en cuestión regresará *cuatro* filas, no dos (aunque tres cuatro filas son idénticas). SQL no elimina las filas redundantes duplicadas a menos que el usuario lo solicite de manera explícita por medio de la palabra clave DISTINCT, como aquí:

```
SELECT DISTINCT P.COLOR, P.CIUDAD
FROM   P
WHERE  P.CIUDAD o 'París'
AND    P.PESO > 10.0 ;
```

Esta consulta devolverá solamente dos filas, no cuatro.

De aquí se sigue que *el objeto de datos fundamental en SQL no es una relación;* más bien es una tabla, y las tablas al estilo de SQL (en general) no contienen conjuntos sino *bolsas* de filas (una "bolsa"—también llamada *multiconjunto*— es como un conjunto pero permite duplicados). Por lo tanto, SQL viola *El principio de información* (vea capítulo 3, sección 3.2). Una consecuencia es que los operadores fundamentales en SQL no son verdaderos operadores relacionales sino equivalentes de éstos; otro aspecto es que los resultados y teoremas que se mantienen verdaderos en el modelo relacional —relacionados, por ejemplo, a la transformación de expresiones [5.6]— no necesariamente se mantienen verdaderos en SQL.

7.7.2 Para todas las partes, obtener el número de parte y el peso de la misma en gramos.

```
SELECT P.P#, P.PESO * 454 AS PSQR
FROM   P ;
```

La especificación AS PSGR introduce un nombre de columna adecuado para la "columna calculada". Entonces, las dos columnas de la tabla de resultado se llaman P# y PSGR, respectivamente. Si se hubiese omitido la cláusula AS, la columna correspondiente en el resultado en efecto no habría tenido nombre. Por lo tanto, observe que en tales circunstancias SQL en realidad no requiere que el usuario proporcione un nombre de columna de resultado, aunque nosotros siempre lo haremos en nuestros ejemplos.

7.7.3 Obtener todas las combinaciones de información de proveedores y partes tales que el proveedor y la parte en cuestión estén coubicados. SQL ofrece muchas formas diferentes de formular esta consulta. Aquí presentamos tres de las más simples.

```
1. SELECT V.*, P.P#, P.PARTE, P.COLOR, P.PESO
   FROM  V, P

   WHERE V.CIUDAD = P.CIUDAD ;

2. V JOIN P USING CIUDAD ;

3. V NATURAL JOIN P ;
```

El resultado en cada caso es la **junta natural** de las tablas V y P (sobre las ciudades).

La primera de las formulaciones anteriores, que es la única que habría sido válida en SQL como se definió originalmente (el soporte explícito de JOIN se incorporó en SQL/92), merece una mejor explicación. De manera conceptual, podemos pensar en esa versión de la consulta como que se implementa de la siguiente forma:

- Primero se ejecuta la cláusula FROM para producir el **producto cartesiano** V TIMES P. (Aquí, deberíamos preocuparnos estrictamente por renombrar las columnas antes de calcular el producto. Ignoramos este aspecto por razones de simplicidad. Además, recuerde que, como vimos en el ejercicio 6.12 en el capítulo 6, el "producto cartesiano" de una sola tabla *T* puede ser considerado como la propia *T* sola.)

- Después se ejecuta la cláusula WHERE para producir una **restricción** de ese productos la que los dos valores de CIUDAD para cada fila son iguales (en otras palabras, ahora hemos calculado la *equijunta* de proveedores y partes sobre las ciudades).
- Por último se ejecuta la cláusula SELECT para producir una **proyección** de esa restricción sobre las columnas especificadas en la cláusula SELECT. El resultado final es la junta natural.

Por lo tanto, en SQL, FROM corresponde al producto cartesiano, WHERE a restringí y SELECT a proyectar (hablando en términos generales), y la formulación SELECT-FROM-WHERE de SQL representa una proyección de una restricción de un producto. Para una explicación, consulte el apéndice A.

7.7.4 Obtener todos los pares de nombres de ciudad tales que un proveedor ubicado en la **primera ciudad suministre una parte almacenada en la segunda ciudad.**

```
SELECT DISTINCT V.CIUDAD AS CIUDADV, P.CIUDAD AS CIUDADP
FROM  V JOIN VP USING V# JOIN P USING P# ¡
```

Observe que lo que sigue *no* es correcto (ya que incluye CIUDAD como columna de junta en la segunda junta):

```
SELECT DISTINCT V.CIUDAD AS CIUDADV, P.CIUDAD AS CIUDADP
FROM  V NATURAL JOIN VP NATURAL JOIN P ;
```

7.7.5 Obtener todos los pares de números de proveedor tales que los **dos** proveedores **involucrados estén coubicados.**

```
SELECT A.V# AS VA, B.V# AS VB
FROM  V AS A, V AS B WHERE
A.CIUDAD ■ B.CIUDAD AND
A.V# < B.V# ;
```

En este ejemplo, se requieren claramente de variables de alcance explícitas. Observe que los nombres de columna introducidos VA y VB se refieren a columnas de la *tabla de resultado y* por ello no pueden ser usados en la cláusula WHERE.

7.7.6 Obtener el número total de proveedores.

```
SELECT COUNT(*   AS N
FROM  V :
```

Aquí, el resultado es una tabla con una columna (llamada N) y una fila que contiene el \
SQL soporta los operadores de totales **COUNT, SUM, AVG, MAX** y **MIN** comunes, auque hay algunos puntos específicos de SQL que el usuario necesita considerar:

- En general, el argumento puede estar precedido de manera opcional por la palabra clave DISTINCT —como en SUM (DISTINCT CANT)— para indicar que se eliminarán los duplicados antes de aplicar la operación. Sin embargo, para MAX y MIN, DISTINC es irre-
  levante y no tiene efecto.
- El operador especial COUNT(*) —no se permite DISTINCT— es proporcionado para contar todas las filas en una tabla sin eliminación alguna de duplicados.

- Todos los nulos en la columna del argumento (vea el capítulo 18) son eliminados antes de efectuar la operación (sin importar si DISTINCT fue especificado), con excepción del caso de COUNT(*), donde los nulos se comportan como si fueran valores.
- Si el argumento resulta ser un conjunto vacío, COUNT regresa cero; todos los demás opera dores regresan nulo. (Este último comportamiento es lógicamente incorrecto —vea la refe rencia [3.3]— aunque es la forma en que está definido SQL).

7.7.7 Obtener las cantidades máxima y mínima de la parte P2.

```
SELECT MAX ( VP.CANT ) AS CMAX, MIN ( VP.CANT ) AS CMIN
FROM   VP
WHERE  VP.P# = 'P2'  ;
```

Observe que aquí ambas cláusulas FROM y WHERE proporcionan efectivamente parte del argumento para los dos operadores de totales. Por lo tanto deberían aparecer lógicamente dentro de los paréntesis que encierran al argumento. No obstante, la consulta se escribe tal como se muestra. Este enfoque poco ortodoxo tiene importantes repercusiones negativas sobre la estructura, la utilización y la ortogonalidad* del lenguaje SQL. Por ejemplo, una consecuencia inmediata es que los operadores de totales no pueden estar anidados, lo que ocasiona que una consulta como "Obtener el promedio de la cantidad total de partes" no pueda ser formulada sin circunlocuciones enredadas. Para ser más específicos, la siguiente consulta es *** *INVÁLIDA* ***:

```
SELECT AVG ( SUM ( VP.CANT ) )                    —¡Advertencia!  ¡Inválida!
FROM   VP ;
```

En su lugar, tiene que ser formulada en forma similar a la siguiente:

```
SELECT AVG ( X )
FROM ( SELECT SUM ( VP.CANT ) AS X
       FROM   VP
       GROUP  BY VP.V# ) AS POINTLESS ;
```

Vea el ejemplo inmediato siguiente para una explicación de GROUP BY, y varios de los ejemplos siguientes para una explicación de las "subconsultas anidadas". Vale la pena señalar que la capacidad para anidar una subconsulta dentro de la cláusula FROM, como en este caso, apareció con SQL/92 y aún no está implementada ampliamente. *Nota:* La especificación AS *POINTLESS* no tiene sentido pero es requerida por las reglas de sintaxis de SQL (vea el apéndice A).

7.7.8 Para cada parte suministrada, obtener el número de parte y la cantidad total del envío.

```
SELECT VP.P#, SUM ( VP.CANT ) AS CANTOT
FROM   VP
GROUP  BY VP.P# ;
```

***Ortogonalidad** significa *independencia.* Un lenguaje es ortogonal si los conceptos independientes se mantienen independientes y no se mezclan en formas confusas. La ortogonalidad es necesaria, ya que entre menos ortogonal sea un lenguaje, resulta más complicado y —de manera paradójica pero simultánea— menos poderoso.

Lo anterior es el equivalente de SQL para la expresión algebraica

```
SUMMARIZE VP PER VP { P# } ADD SUM ( CANT ) AS CANTOT
```

o para la expresión del cálculo de tupias

```
( VPX.P#, SUM ( VPY WHERE VPY.P# ■ VPX.P#, CANT ) AS CANTOT )
```

En particular observe que si la cláusula GROUP BY es especificada, las expresiones en la cláusula SELECT deben tener **un solo valor por grupo.**

Aquí tenemos una formulación alternativa (y de hecho, preferible) de la misma consulta:

```
SELECT P.P#, ( SELECT SUM ( VP.CANT )
               FROM  VP
               WHERE VP.P# = P.P# ) AS CANTOT
FROM P ;
```

La capacidad para usar subconsultas anidadas para representar valores escalares (por ejemplo, dentro de la cláusula SELECT, como aquí) fue incorporada en SQL/92 y representa una mejora importante sobre el SQL definido originalmente. En el ejemplo, esto nos permite generar un resultado que incluye filas de partes que no son suministradas en lo absoluto, lo que no hace la formulación anterior (usando GROUP BY). Sin embargo, el valor CANTOT para dichas partes se dará desafortunadamente como nulo, no como cero.

7.7.9 Obtener los números de partes suministradas por más de un proveedor.

```
SELECT VP.P#
FROM  VP
GROUP BY VP.P#
HAVING COUNT ( VP.V# ) > 1 ;
```

La cláusula HAVING es a los grupos lo que la cláusula WHERE es a las filas; en otras palabra HAVING se usa para eliminar grupos, tal como WHERE se usa para eliminar filas. Las expresiones en una cláusula HAVING deben tener un solo valor por grupo.

7.7.10 Obtener los nombres de los proveedores que suministran la parte P2.

```
SELECT DISTINCT V.PROVEEDOR
FROM  V WHERE V.V# IN
     ( SELECT VP.V# FROM  VP
       WHERE VP.P# = 'P2' ) ;
```

Explicación: Este ejemplo hace uso de una subconsulta en la cláusula WHERE. A grandes ras-gos, una **subconsulta** es una expresión SELECT-FROM-WHERE-GROUP BY-HAVING que está anidada en alguna parte dentro de otra de estas expresiones. Las subconsultas se usan, **entre** otras cosas, para representar el conjunto de valores a examinar mediante una condición IN, **como** ilustra el ejemplo. El sistema evalúa la consulta general evaluando primero la subconsulta (por lo menos de manera conceptual). Esa subconsulta regresa el conjunto de *números* de los proveedores que suministran la parte P2; para ser más específicos, el conjunto {Vl,V2,V3,V4).Por tanto, la expresión original es equivalente a la siguiente expresión más sencilla:

```
SELECT DISTINCT V.PROVEEDOR
FROM   V
WHERE  V.V# IN ( 'V1', 'V2', 'V3', 'V4' )
```

Vale la pena destacar que el problema original —"Obtener los nombres de los proveedores que suministran la parte P2"— puede formularse igualmente bien por medio de *una junta,* como se muestra a continuación:

```
SELECT DISTINCT V.PROVEEDOR
FROM  V, VP
WHERE V.V# = VP.V#
AND   VP.P# = 'P2' ;
```

7.7.11 Obtener los nombres de los proveedores que suministran por lo menos una parte roja.

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHERE V.V# IN
   ( SELECT  VP.V#
     FROM    VP
     WHERE   VP.P#  I
          ( SELEC  P ,P0
            FROM   P
            WHERE  P .COLOR
                                'Rojo' ) ) ;
```

Las subconsultas pueden anidarse en cualquier nivel de profundidad. *Ejercicio:* Proponga algunas formulaciones equivalentes de junta para esta consulta.

7.7.12 Obtener los números de los proveedores con un status menor que el status máximo actual de la tabla V.

```
SELECT V.V#
FROM  V
WHERE V.STATUS <
     ( SELECT MAX ( V.STATUS )
       FROM  V ) ;
```

Este ejemplo comprende *dos variables de alcance implícitas distintas,* ambas denotadas por el mismo símbolo "V" y ambas abarcando la tabla V.

7.7.13 Obtener los nombres de los proveedores que suministran la parte P2. *Nota:* Éste es el mismo ejemplo que 7.7.10, pero mostramos una solución diferente con el fin de presentar otra característica de SQL.

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHER  EXISTS
   ( SELECT  *
     FROM    VP
     WHERE   VP.V# ,
     AND     VP.P# = 'P2'
```

Explicación: La expresión SQL "EXISTS (SELECT ... FROM ...)" da como resultado *verdadero* si, y sólo si, el resultado de evaluar "SELECT... FROM..." no está vacío. En otras palabras, el *operador* **EXISTS** de SQL corresponde al *cuantificador existencial* del cálculo de tupias (pero vea la referencia [18.6]). *Nota:* En este ejemplo en particular, SQL se refiere a la subconsulta como una subconsulta *correlacionada,* ya que incluye referencias a una variable de alcance

—es decir, la variable de alcance implícita V— que está definida en la consulta exterior. Para otro ejemplo de una subconsulta correlacionada, consulte de nuevo la "formulación preferible del ejemplo 7.7.8.

7.7.14 Obtener los nombres de aquellos proveedores que no suministran la parte P2.

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHERE NOT EXISTS
     ( SELECT *
       FROM  VP
       WHERE  VP.V# = V.V#
       AND   VP.P# = 'P2' ) ;
```

Como alternativa:

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHERE V.V# NOT IN ( SELECT
     VP.V# FROM  VP WHERE
     VP.P# = 'P2' ) ;
```

7.7.15 Obtener los nombres de los proveedores que suministran todas las partes.

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHERE NOT EXISTS
     ( SELECT •
     FROM  P
       WHERE NOT EXISTS (
            SELECT * FROM
            VP
              WHERE VP.V# ■ V.V# AND
              VP.P# = P.P# ) ) ;
```

SQL no incluye ningún soporte directo del cuantificador universal FORALL; de ahí que. por lo regular, las consultas "FORALL" tengan que ser expresadas en términos de cuantifícadores exis-tenciales y doble negación, como en este ejemplo.

Por cierto, vale la pena señalar que aunque expresiones como la que acabamos de me podrían parecer a primera vista desalentadoras, un usuario familiarizado con el cálculo relaciona puede construirlas con facilidad, como explica la referencia [7.4]. Como alternativa —si aún son demasiado desalentadoras— entonces existen varios enfoques para "darles la vuelta", los pueden ser empleados para evitar la necesidad de cuantificadores negados. En el ejemp dríamos escribir:

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHERE ( SELECT  COUNT (
        FROM    VP
        WHERE   VP.V# =
      ( SELECT  COUNT (
        FROM P
```

("Obtener los nombres de los proveedores donde la cuenta de las partes que suministran sea igual a la cuenta de todas las partes".) Sin embargo, observe que:

- Primero, esta última formulación depende —al contrario de la formulación NOT EXISTS— del hecho de que todo número de parte de un envío es el número de alguna parte existente. En otras palabras, las dos formulaciones son equivalentes (y la segunda es correcta) sólo porque está en efecto cierta *restricción de integridad* (vea el siguiente capítulo).
- Segundo, la técnica utilizada en la segunda formulación para comparar las dos cuentas (las subconsultas en ambos lados del signo de igual) no era soportada en SQL tal como fue definido originalmente, sino que fue agregada en el SQL/92. Muchos productos aún no la soportan.
- También subrayamos que lo que en realidad nos gustaría hacer es comparar dos *tablas* (vea la explicación de las comparaciones relacionales en el capítulo 6), para expresar entonces la consulta como sigue:

```
SELECT DISTINCT V.PROVEEDOR
FROM  V
WHERE  ( SELECT  VP.P#
         FROM    VP
         WHERE   VP.V# = V.V# ) =
       ( SELECT P.P#
         FROM P ) ;
```

  Sin embargo, SQL no soporta directamente las comparaciones entre tablas y por ello tenemos que recurrir al truco de comparar las *cardinalidades* de las tablas (dependiendo de nuestro propio conocimiento externo para asegurar que si las cardinalidades son las mismas, entonces las tablas son también las mismas, por lo menos en la situación que nos ocupa). Vea el ejercicio 7.11 al final del capítulo.

7.7.16 Obtener los números de partes que pesan más de 16 libras o que son suministradas por el proveedor V2, o ambas cosas.

```
SELECT P.P#
FROM  P
WHERE P.PESO > 16.0

UNION

SELECT VP.P#
FROM  VP
WHERE VP.V# = 'V2' ;
```

Las filas duplicadas redundantes siempre son eliminadas del resultado de **UNION, INTERSECT** o **EXCEPT** no calificados (EXCEPT es el equivalente de SQL para nuestro MINUS). Sin embargo, SQL también ofrece las variantes calificadas **UNION ALL, INTERSECT ALL y EXCEPT ALL,** donde los duplicados (si los hay) son conservados. De manera deliberada omitimos ejemplos de estas variantes.

Esto nos lleva al final de nuestra lista de ejemplos de recuperación. La lista es más bien larga; sin embargo, existen diversas características de SQL que ni siquiera hemos mencionado. El hecho es que SQL es un lenguaje sumamente *redundante* [4.18], en el sentido de que casi siempre ofrece diversas maneras de formular una misma consulta; y el espacio del que aquí disponemos sencillamente no nos permite describir todas las formulaciones y todas las opciones posibles, incluso para el número de ejemplos comparativamente reducido que expusimos en esta sección.

7.8 RESUMEN

Hemos considerado brevemente el **cálculo relational,** como una alternativa al álgebra cional. De manera superficial, ambos lucen muy diferentes —el cálculo es **descriptivo** mientras que el álgebra es **prescriptiva**—, pero en el fondo son lo mismo, ya que cualquier expresión del cálculo puede ser convertida en una expresión semánticamente equivalente del álgebra y vice-*versa.*

El cálculo se presenta en dos versiones, cálculo de **tupias** y cálculo de **dominios.** La dife-rencia clave entre ellos es que las variables de alcance del cálculo de tupias abarcan relacione mientras que las variables de alcance del cálculo de dominios abarcan dominios.

Una expresión del cálculo de tupias consiste en una **prototupla** y una cláusula WHERE cional que contiene una expresión lógica o **WFF** (fórmula bien formada). Se permite que esa WFF contenga **cuantificadores** (EXISTS y FORALL), **referencias a variable libres** y ligadas, operadores lógicos (AND, OR, NOT, etcétera), y así sucesivamente. Toda variable libre cionada en la WFF también debe ser mencionada en la prototupla. *Nota:* No expusimos de mane- ra explícita esta idea en el cuerpo del capítulo, pero las expresiones del cálculo están diseñadas para servir básicamente a los mismos fines que las expresiones del álgebra (vea el capítol sección 6.6).

Mostramos mediante un ejemplo cómo puede ser usado el **algoritmo de reducción** de Codd para convertir una expresión cualquiera de cálculo a una expresión equivalente de álgebra, allí nando así el camino para una posible estrategia de implementación del cálculo. Mencionan una vez más el aspecto de la **compleción relacional** y explicamos brevemente lo que involucra la demostración de que un lenguaje dado *L* sea completo en este sentido.

También consideramos la cuestión de incluir posibilidades **computacionales** (análogas las capacidades que proporcionan EXTEND y SUMMARIZE en el álgebra) en el cálculo de tu-plas. Después presentamos una breve introducción al cálculo de **dominios** y afirmamos (sin in-tentar demostrarlo) que éste era también relacionalmente completo. Por lo tanto, el cálculo de tupias, el cálculo de dominios y el álgebra son todos equivalentes entre sí.

Por último, presentamos un repaso de las características relevantes de **SQL.** SQL es una es-pecie de híbrido entre el álgebra y el cálculo (de tupias); por ejemplo, incluye soporte din para los operadores JOIN y UNION del álgebra, pero además utiliza las variables de alcana el cuantificador existencial del cálculo. Una consulta de SQL consiste en una **expresión** de tabla. Por lo regular, esta operación consiste en una sola **expresión de selección,** pero también son so-portadas varias clases de expresiones **JOIN** explícitas y es posible combinar expresiones de junta y selección de distintas formas utilizando los operadores **UNION, INTERSECT** y EXCE También mencionamos el uso de **ORDER BY** para imponer un orden en la tabla resultan una expresión de tabla (de cualquier clase).

Con respecto a las **expresiones de selección** en particular, describimos:

- La **cláusula SELECT** básica, que incluye el uso de **DISTINCT,** expresiones escalares, la
  introducción de nombres de columnas de resultado y "SELECT *";
- La **cláusula FROM,** que incluye el uso de **variables de alcance;**
- La **cláusula WHERE,** que incluye el uso del operador **EXISTS;**

- Las **cláusulas GROUP BY y HAVING,** que incluyen el uso de los **operadores de totales** COUNT, SUM, AVG, MAX y MIN;
- El uso de **subconsultas** en (por ejemplo) las cláusulas SELECT, FROM y WHERE.

También proporcionamos un **algoritmo de evaluación conceptual** —es decir, un esbozo de una definición formal— para las expresiones de selección.

EJERCICIOS

7.1 Sean *p(x)* y *q* WFFs cualesquiera en las cuales *x* aparece y no aparece, respectivamente, como una variable libre. ¿Cuáles de las siguientes instrucciones son válidas? (El símbolo "=►" significa "implica"; el símbolo "=" significa "es equivalente a". Observe que "A =► *B"* y *"B* =► A" son lo mismo que "A = S.")

a. EXISTS x (*q*) = *q*

b. FORALL x (*q)* = *q*

C. EXISTS x (p(x) AND *q)* = EXISTS x (p(x)) AND *q*

d. FORALL x (p(x) AND *q*) = FORALL x (p(x)) AND *q*

e. FORALL x (p(x)) =► EXISTS x (p(x))

7.2 Sea *p(x,y)* una WFF cualquiera con las variables libres xy y. ¿Cuáles de las siguientes instruc ciones son válidas?

a. EXISTS x EXISTS y (*p(x,y)*) = EXISTS y EXISTS x (*p(x,y)*)

b. FORALL x FORALL y (*p{x,y))* = FORALL y FORALL x (*p(x,y))*

C. FORALL x (*p(x,y))* = NOT EXISTS x (NOT p(x,y))

d. EXISTS x (p(x,y)) = NOT FORALL x (NOT *p(x,y))*

e. EXISTS x FORALL y (*p(x,y))* = FORALL y EXISTS x (p(x,y))

f. EXISTS y FORALL x (*p(x,y))* =► FORALL x EXISTS y (p(x,y))

7.3 Sean *p(x)* y *q(y)* WFFs cualesquiera con las variables libres *x* y y, respectivamente. ¿Cuáles de las siguientes instrucciones son válidas?

a. EXISTS x (p(x)) AND EXISTS y (q(y)) =
EXISTS x EXISTS y (p(x) AND *q{y))*

b. EXISTS x (IF p(x) THEN q(x) END IF) ■
IF FORALL X (p(x)) THEN EXISTS X (p(x)) END IF

7.4 Considere una vez más el ejemplo 7.3.8 —"Obtener los números de los proveedores que su ministran por lo menos todas las partes que suministra el proveedor V2"— para la cual una posible formulación del cálculo de tupias es

```
VX.V# WHERE FORALL VPY ( IF VPY.V# = V# ( 'V2' ) THEN
                            EXISTS VPZ ( VPZ.V# = VX.V# AND
                                         VPZ.P# = VPY.P# )
                         END IF )
```

(Aquí, VPZ es otra variable de alcance que abarca los envíos.) ¿Qué devolverá esta consulta si proveedor V2 no suministra actualmente ninguna parte? ¿Cuál sería la diferencia si en todos los lu gares remplazáramos VX por VPX?

7.5 Aquí tenemos una consulta de ejemplo tomada de la base de datos de proveedores, paita] proyectos (se aplican las convenciones usuales con respecto a los nombres de variables de alcance

```
( PX.PARTE, PX.CIUDAD ) WHERE FORALL VX FORALL YX EXISTS VPYX
                                    ( VX.CIUDAD = 'Londres' AND
                                      YX.CIUDAD = 'París' AND
                                      VPYX.V# = VX.V# AND VPYX.P#
                                      = PX.P# AND VPYX.Y# = YX.Y#
                                      AND VPYX. CANT < CANT ( 500 )
                                      )
```

a. Traduzca esta consulta al lenguaje natural.

b. Tome el papel del DBMS y "ejecute" el algoritmo de reducción de Codd sobre esta consulta.
   ¿Puede detectar alguna mejora posible para ese algoritmo?

7.6 Proponga una formulación del cálculo de tupias de la consulta "Obtener las tres partes mas pesadas".

7.7 Considere la varrel de *lista de materiales* ESTRUCTURA_PARTES del capítulo 4 (damos una definición de SQL en la respuesta al ejercicio 4.6 y presentamos un ejemplo de valor de relación en la figura 4.6). La bien conocida consulta de explosión de partes "Obtener los números de todas las partes que son componentes *a cualquier nivel* de alguna parte dada, digamos la parte Pl"—el resultado de la cual, digamos LISTA_PARTES, es en realidad una relación que puede derivarse de ESTRUC-TURA_PARTES— no puede ser formulada como una sola expresión del cálculo relacional (o álgebra) original. En otras palabras, LISTA_PARTES es una relación derivable que sin embargo no puese ser derivada por medio de una sola expresión del cálculo (o álgebra) original. ¿Por qué sucede esto?

7.8 Suponga que la varrel de proveedores V fuera sustituida por un conjunto de varrels LV,P1 AV... (una para cada ciudad de proveedor distinta; por ejemplo, la varrel LV contendría sólo las tuplas de los proveedores en Londres). Suponga también que no estamos al tanto de qué ciudades de provee-dores existen y por lo tanto desconocemos exactamente cuántas varrels hay. Considere la con "¿Está representado el proveedor V1 en la base de datos?" ¿Se puede expresar esta consulta en el cálcu-lo (o en el álgebra)? Justifique su respuesta.

7.9 Demuestre que SQL es relacionalmente completo.

7.10 ¿Tiene SQL equivalentes de los operadores relacionales EXTEND y SUMMARIZE? j

7.11 ¿Tiene SQL equivalentes de los operadores relacionales de comparación?

7.12 Proponga tantas formulaciones de SQL como pueda imaginar para la consulta "Obtener los nombres de los proveedores que suministran la parte P2".

Ejercicios de consultas

Todos los ejercicios restantes están basados en la base de datos de proveedores, partes y proyectos (vea la figura 4.5 en la sección de "Ejercicios" del capítulo 4 y la respuesta al ejercicio 5.4 del capitulo 5). En cada caso le pedimos que escriba una expresión para la consulta indicada. (Como una vavariante

interesante, podría intentar ver primero algunas de las respuestas y establecer lo que significa en el lenguaje natural la expresión dada.)

7.13 Proponga soluciones de cálculo de tupias para los ejercicios 6.13 al 6.50.

7.14 Proponga soluciones de cálculo de dominios para los ejercicios 6.13 al 6.50.

7.15 Proponga soluciones de SQL para los ejercicios 6.13 al 6.50.

REFERENCIAS Y BIBLIOGRAFÍA

Además de las siguientes, también consulte la referencia [4.7], que describe algunas extensiones a SQL para tratar el cierre transitivo y aspectos similares. SQL3 incluye otras extensiones similares, las cuales describimos brevemente en el apéndice B. Con respecto a la incorporación de dichas propiedades dentro del cálculo relacional como tal, consulte el capítulo 23.

7.1 E. F. Codd: "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif, (noviembre, 1971).

7.2 C. J. Date: "A Note on the Relational Calculus", *ACM SIGMOD Record 18*, No. 4 (diciembre, 1989). Reeditado como "An Anomaly in Codd's Reduction Algorithm" en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

7.3 C. J. Date: "Why Quantifier Order Is Important", en C. J. Date y Hugh Darwen, *Relational Data base Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

7.4 C. J. Date: "Relational Calculus as an Aid to Effective Query Formulation", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

En la actualidad, prácticamente todos los productos del mercado soportan SQL, aunque no el cálculo o el álgebra relacionales. No obstante, este documento ilustra y aboga por el uso del cálculo relacional como un paso intermedio en la construcción de consultas "complejas" de SQL.

7.5 G. D. Held, M. R. Stonebraker y E. Wong: "INGRES—A Relational Data Base System", Proc. *NCC 44*, Anaheim, Calif. Montvale, N.J.: AFIPS Press (mayo, 1975).

A mediados de los años setenta existían dos grandes prototipos relacionales en desarrollo: El System R en IBM e Ingres (originalmente INGRES, en mayúsculas), en la Universidad de California en Berkeley. Estos dos proyectos se hicieron sumamente influyentes en el mundo de la investigación y ambos condujeron subsecuentemente a sistemas comerciales, incluyendo a DB2 en el caso del System R y del producto comercial Ingres. *Nota:* Al prototipo de Ingres, en ocasiones se le llama "University Ingres" [7.11] a fin de distinguirlo de la versión comercial del sistema, "Commercial Ingres". Puede encontrar un repaso tutorial de la versión comercial en la referencia [1.5]. Originalmente, Ingres no fue un sistema SQL; en su lugar, soportaba un lenguaje denominado QUEL ("Query Language"), que en muchos aspectos era técnicamente superior a SQL. De hecho, QUEL sigue conformando la base de una cierta cantidad de la investigación en bases de datos y en la literatura de investigación aún aparecen ejemplos expresados en QUEL. Este artículo, que fue el primero en describir el prototipo Ingres, incluye una definición preliminar de QUEL. También vea las referencias [7.10 a 7.12].

7.6 J. L. Kuhns: "Answering Questions by Computer: A Logical Study", Report RM-5428-PR, Rand Corp., Santa Monica, Calif. (1967).

7.7 M. Lacroix y A. Pirotte: "Domain-Oriented Relational Languages", Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japan (octubre, 1977).

7.8 T. H. Merrett: "The Extended Relational Algebra, A Basis for Query Languages", en B. Sh derman (ed.), *Databases: Improving Usability and Responsiveness.* New York, N.Y.: Academic I (1978).

Propone la introducción de cuantificadores al álgebra *[sic]* —no sólo los cuantificadores existencial y universal del cálculo, sino los cuantificadores más generales "el número de" y "la proporción de" (por ejemplo, "por lo menos tres de", "no más de la mitad de", "un número non de", etcétera).

7.9 M. Negri, G. Pelagatti y L. Sbattella: "Formal Semantics of SQL Queries", *ACM TODS 16,* Nº. 3 (septiembre, 1991).

Citando del resumen: "La semántica de las consultas de SQL se define formalmente estableciendo una serie de reglas que determinan una traducción conducida por sintaxis de una consulta SQL, a un modelo formal denominado E3VPC (Extended Three-Valued Predicate Calculus), el cual se basa en gran medida en conceptos matemáticos bien conocidos. También se dan reglas para transformar una expresión E3 VPC general a una forma canónica; [además] se resuelven por completo problemas como el análisis de equivalencias de consultas de SQL". Sin embargo, tenga presente que el dialecto de SQL considerado en el texto es sólo la primera versión del estándar ("SQL/86"), no SQL/92.

7.10 Michael Stonebraker (ed.): *The INGRES Papers: The Anatomy of a Relational Database Man agement System.* Reading, Mass.: Addison-Wesley (1986).

Una colección de algunos de los principales artículos del proyecto University Ingres, editado y comentado por uno de los diseñadores originales de Ingres. (Las referencias [7.11 y 7.12] están incluidas en la colección.) A saber de este autor, éste es el único libro disponible que describe en detalle el diseño e implementatión de un DBMS relacional a gran escala. Una lectura esencial para el estudiante serio.

7.11 Michael Stonebraker, Eugene Wong, Peter Kreps y Gerald Held: "The Design and Implementa tion of INGRES", *ACM TODS I,* No. 3 (septiembre, 1976). Reeditado en la referencia [7.10].

Una descripción detallada del prototipo University Ingres.

7.12 Michael Stonebraker: "Retrospection on a Data Base System", *ACM TODS 5,* No. 2 (junio, 1980). Reeditado en la referencia [7.10].

Un recuento de la historia del proyecto del prototipo Ingres (hasta enero de 1979). Más que en los éxitos, se pone énfasis en los errores y las lecciones aprendidas.

7.13 Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems: Volume I.* Rockville, Md.: Computer Science Press (1988).

El libro de Ullman comprende un tratamiento más formal del cálculo relacional y de los aspectos afines que el que realiza el presente libro. En particular, expone el concepto de la **seguridad de** las expresiones del cálculo. Éste es un tema importante si adoptamos una versión ligeramente diferente del cálculo; una en la cual las variables de alcance no estén definidas por instrucciones separadas, sino que en vez de ello estén ligadas a su alcance por medio de expresiones explícitas dentro de la cláusula WHERE. En una versión así del cálculo, la consulta (por ejemplo) "Obtener los proveedores en Londres" podría lucir similar a ésta:

```
MX                    VX.CIUOMJ =
                      'Londres'
```

Un problema (no el único) con esta versión del cálculo es que permitiría aparentemente una consulta como

```
VX WHERE NOT ( VX e V )
```

Decimos que tal expresión "no es segura", ya que no devuelve una respuesta finita (el conjunto de todas las cosas que no son tupias de V es infinito). Como consecuencia, es necesario imponer ciertas reglas para garantizar que todas las expresiones válidas sean seguras. Dichas reglas se exponen en el libro de Ullman (tanto para el cálculo de tupias como para el de dominios). Nosotros señalamos que el cálculo original de Codd sí incluía esas reglas.

7.14 Moshé M. Zloof: "Query-By-Example", Proc. NCC *44,* Anaheim, Calif, (mayo, 1975). Montvale, N.J.: AFIPS Press (1977).

El lenguaje relacional QBE **(Query-By-Example)** incorpora elementos tanto del cálculo de tupias como del de dominios (con énfasis en el segundo). Su sintaxis, que es atractiva e intuitivamente muy sencilla, se basa en la idea de hacer entradas en "tablas esqueleto". Por ejemplo, una formulación de QBE para la consulta "Obtener los nombres de proveedores que suministran por lo menos una de las partes que suministra el proveedor V2" (una consulta bastante compleja) podría lucir como ésta:

V#	PROVEEDOR	VP	V#	p#	VP	V#	P#
VX	P. NX		VX	PX		V2	PX

Explicación: El usuario pide al sistema que muestre en la pantalla tres esqueletos de tabla, uno para proveedores y dos para envíos, y posteriormente realiza entradas en ellos como se muestra. Las entradas que comienzan con un carácter de subrayado son "ejemplos" (es decir, variables de alcance para cálculo de dominios); otras entradas son valores literales. El usuario está solicitando al sistema que "presente" ("P.") valores de nombre de proveedor (_NX) tales que si el proveedor es _VX, entonces _VX suministre alguna parte _PX y a su vez, la parte _PX sea también suministrada por el proveedor V2. Observe que los cuantificadores existenciales están implícitos (como, por cierto, lo están también en QUEL); otra razón por la que la sintaxis es intuitivamente fácil de entender.

Aquí tenemos otro ejemplo: "Obtener los pares de números de proveedor tales que los proveedores involucrados estén coubicados".

V#	CIUDAD
VX	_CZ
VY	CZ

P.	VX	VY

Por desgracia, QBE no es relacionalmente completo. Para ser específicos, no maneja adecuadamente el cuantificador existencial negado (NOT EXISTS). Como resultado, ciertas consultas (por ejemplo, "Obtener los nombres de los proveedores que suministran todas las partes") no pueden expresarse en QBE. (En realidad, QBE sí "soportaba" originalmente a NOT EXISTS, por lo menos de manera implícita, pero la construcción siempre fue de algún modo problemática. El problema básico fue que no había forma de especificar el orden en que los diferentes cuantificadores implícitos iban a ser aplicados y por desgracia, el orden es importante; vea la referencia [7.3] o la respuesta al ejercicio 7.2. En consecuencia, ciertas formulaciones de consultas QBE eran ambiguas [7.3].)

Zloof fue el inventor y diseñador original de QBE. Este artículo fue el primero de muchos escritos por Zloof sobre el tema.

RESPUESTAS A EJERCICIOS SELECCIONADOS

7.1 a. Válida, b. Válida, c. Válida, d. Válida, e. Inválida. *Nota:* La razón por la que e. no es válida es que FORALL aplicado a un conjunto vacío produce *verdadero,* mientras que EXISTS aplicado a un conjunto vacío produce *falso.* Así que, por ejemplo, el hecho de que la proposición "Todas las partes moradas pesan más de 100 libras" sea *verdadera* (es decir, es una proposición verdadera) no necesariamente significa que en realidad existan partes moradas.

Subrayamos que las equivalencias e implicaciones (j válidas!) pueden ser usadas como base para un conjunto de reglas de transformación de expresiones del cálculo; muy parecidas a las reglas transformación de expresiones algebraicas que mencionamos en el capítulo 6 y que explicamos en talle en el capítulo 17. Una observación similar también se aplica a las respuestas de los ejercicios' y 7.3.

7.2 a. Válida, b. Válida, c. Válida (ésta la explicamos en el cuerpo del capítulo), d. Válida (c que cada uno de los cuantificadores pueda ser definido en términos del otro), e. Inválida, f. Válida Observe que (como muestran a. y b.) es posible escribir una secuencia de cuantificadores iguales cualquier orden sin cambiar el significado, mientras que (como muestra e.) para los cuantificadores diferentes el orden es importante. A manera de ejemplo de este último punto, abarquen x y y en con-
junto de enteros y sea p la WFF *"y > x"*. Debe quedar claro que la WIT

```
FORALL x EXISTS y ( y > x )
```

("Para todos los enteros *x*, existe un entero más grande *y")* da como resultado *verdadero*, en tanto que la WFF

```
EXISTS y FORALL x ( y > x )
```

("Existe un entero y que es más grande que todo entero *x")* da como resultado *falso*. De ahí que in-tercambiar cuantificadores diferentes cambie el significado. Por lo tanto, en un lenguaje de consul basado en el cálculo, intercambiar cuantificadores diferentes en una cláusula WHERE cambiará el i significado de la consulta. Vea la referencia [7.3].

7.3 a. Válida, b. Válida.

7.4 Si el proveedor V2 no suministra actualmente parte alguna, la consulta original devolverá todos
los números de proveedor que aparecen en V (incluyendo en particular al proveedor V2, quien sumamente aparece en V pero no en VP). Si reemplazamos todo VX por VPX, obtendremos tod los números de proveedor que aparecen actualmente en VP. La diferencia entre las dos formulaciones es entonces la siguiente. La primera significa "Obtener los números de los proveedores que suminis-tran por lo menos todas las partes que suministra el proveedor V2" (como fue requerido). La segunda significa "Obtener los números de los proveedores que *suministran por lo menos una parle* y sumi-
nistran por lo menos todas las partes que suministra el proveedor V2".

7.5 a. Obtener el nombre de parte y la ciudad de las partes suministradas por cada proveedor de Lon-
dres a cada proyecto en París, en una cantidad menor a 500. b. El resultado de esta consulta está vacío

7.6 ¡Este ejercicio es muy difícil! En especial cuando tomamos en cuenta el hecho de que los pesos
de las partes no son únicos. (Si lo fueran, podríamos parafrasear la consulta como "Obtener tod partes tales que la cuenta de las partes más pesadas sea menor que tres"). De hecho, el ejercicio es tan
difícil, que aquí no intentamos dar una solución de cálculo puro. Éste ilustra muy bien la idea de que la compleción relacional es sólo una medida *básica* de poder expresivo, no necesariamente una sufi-
ciente. (Los dos ejercicios que siguen también ilustran esta idea.) Para una explicación de consultas de este tipo, vea la referencia [6.4].

7.7 Sean PVA, PVB, PVC, ..., PVn variables de alcance que abarquen (el valor actual de la varia-
ble de relación ESTRUCTURA_PARTES, y suponga que la parte dada es P1. Entonces:

a. Una expresión del cálculo para la consulta "Obtener los números de parte de todas las partes que son componentes, en el *primer* nivel, de la parte Pl" es:

   ```
   PVA.P#_MENOR WHERE PVA.P#_MAYOR = P# ( ' Pl ' )
   ```

b. Una expresión del cálculo para la consulta "Obtener los números de parte de todas las partes son componentes, en el *segundo* nivel, de la parte Pl" es:

```
PVB. P# _MENOR WHERE EXISTS PVA
                    ( PVA.P#_MAYOR = P# ( 'P1' ) AND
                      PVB.P#_MAYOR - PVA.P#_MENOR )
```

c. Una expresión del cálculo para la consulta "Obtener los números de parte de todas las partes que son componentes, en el *tercer* nivel, de la parte Pl" es:

```
PVC.P#_MENOR WHERE EXISTS PVA EXISTS PVB
                    ( PVA.P#_MAYOR = P# ( 'PV ) AND
                      PVB.P#_MAYOR = PVA.P#_MENOR AND
                      PVC.P#_MAYOR » PVB.P#_MENOR )
```

Y así sucesivamente. Una expresión del cálculo para la consulta "Obtener los números de parte de todas las partes que son componentes, en el *enésimo* nivel, de la parte Pl" es:

```
PVn.P#_MENOR WHERE EXISTS PVA EXISTS PVB ... EXISTS PV(n-1) (
                    PVA.P#_MAYOR = P# ( 'P1[1] ) AND PVB. P#_MAYOR
                    - PVA.P#_MENOR AND PVC.P#_MAYOR =
                    PVB.P#_MENOR AND
                          ..............................  AND
                      PVn.P#_MAYOR ■ PV(n-1).P#_MENOR )
```

Todas estas relaciones resultantes a., b., c, etcétera, necesitan entonces "unirse" para construir el resultado LISTA_PARTES.

Por supuesto, el problema es que no hay forma de escribir éstas *n* expresiones si se desconoce el valor de *n.* De hecho, la consulta de explosión de partes es una ilustración clásica de un problema que no puede ser formulado por medio de una sola expresión en un lenguaje que sólo es completo relacionalmente; es decir, un lenguaje que no es más poderoso que el cálculo (o el álgebra) original. Por lo tanto, necesitamos otra extensión para el cálculo (y el álgebra) original. El operador TCLOSE que explicamos brevemente en el capítulo 6, es parte de la solución a este problema (pero sólo parte). Más detalles están fuera del alcance de este libro.

Nota: Aunque a este problema se le conoce por lo regular como "lista de materiales" o "explosión de partes", en realidad tiene una aplicabilidad mucho más amplia de lo que podrían sugerir esos nombres. De hecho, la clase de relación tipificada por la estructura "las partes contienen partes" ocurre en un amplio rango de aplicaciones. Otros ejemplos incluyen las jerarquías de administración, los árboles genealógicos, los grafos de autorización, las redes de comunicaciones, las estructuras de invocación a módulos de software, las redes de transporte, etcétera.

7.8 Esta consulta no puede ser expresada en el cálculo ni en el álgebra. Por ejemplo, para expresarla en el cálculo necesitaríamos básicamente poder decir algo como lo siguiente:

¿Existe una relación *r* tal que exista una tupia *t* en *r* tal que t.V# = V#('V1')?

En otras palabras, tendríamos que poder cuantificar sobre *relaciones* en lugar de sobre tupias, y por lo tanto necesitaríamos una nueva clase de variable de alcance que denote relaciones en lugar de tupias. Por lo tanto, la consulta no puede ser formulada en el cálculo relacional tal como está definido actualmente.

Por cierto, observe que la consulta en cuestión es una consulta "sí/no" (la respuesta deseada es básicamente un valor verdadero). Por ello tal vez se vea tentado a pensar que la razón por la que la consulta no puede ser manejada en el cálculo ni en el álgebra es que las expresiones en éstos tienen un valor de relación, no un valor de verdad. Sin embargo, las consultas sí/no pueden ser manejadas en el cálculo y en el álgebra si se implementan en la forma adecuada. El meollo del asunto está en reconocer que el sí y el no (en forma equivalente, *verdadero* y *falso)* pueden *ser representados como relaciones.* Las relaciones en cuestión son TABLE_DEE y TABLE_DUM, respectivamente [5.5].

7.9 A fin de mostrar que SQL es relacionalmente completo, primero tenemos que mostrar que exis ten expresiones de SQL para cada uno de los cinco operadores —algebraicos— primitivos (restringir, proyectar, producto, unión y diferencia) y después que los operandos para esas expresiones de **SQL** pueden a su vez ser expresiones cualesquiera de SQL.

Comenzamos con la observación de que SQL soporta en efecto el operador RENAME del álge bra relacional, gracias a la introducción en SQL/92 de la especificación opcional "AS <*nombre de columna*>" sobre elementos de la cláusula SELECT. Entonces, podemos asegurar que todas las tablas sí tienen nombres de columna adecuados, y en particular que los operandos para producto, unión y diferencia satisfacen los requerimientos de (nuestra versión de) el álgebra con respecto a la denomi-nación de las columnas. Además, —siempre y cuando dichos requerimientos de denominación de columnas de operandos sean en realidad satisfechos— las reglas de herencia de nombres de columna de SQL coinciden de hecho con las del álgebra tal como las describimos (bajo el nombre de inferen-*cia del tipo de relación)* en el capítulo 6.

Entonces, aquí están las expresiones de SQL correspondientes a los cinco operadores primitiva

Álgebra	*SQL*
	`SELECT * FROM A WHERE p`
	`SELECT DISTINCT x,y,...,z FROM A`
	`A CROSS JOIN B`
	`SELECT * FROM A UNION SELECT * FROM B`
	`SELECT * FROM A EXCEPT SELECT * FROM B`

Haciendo referencia a la gramática de las expresiones de tabla de SQL dadas en el apéndice A vemos que cada *A* y *B* en las expresiones de SQL mostradas arriba es de hecho una <*referencia tabla*>. También vemos que si tomamos cualquiera de las cinco expresiones SQL mostradas y encerramos entre paréntesis, lo que a su vez resulta es una <*referencia a tabla*>.* Se desprende que SQL es en realidad relacionalmente completo.

Nota: De hecho en lo anterior existe un error; SQL no maneja la *proyección sobre ninguna columna en absoluto* (debido a que no permite cláusulas SELECT vacías). Como consecuencia, no maneja TABLE_DEE ni TABLE_DUM [5.5].

7.10 SQL soporta EXTEND pero no SUMMARIZE (por lo menos no muy directamente). Con respecto a EXTEND, la expresión del álgebra relacional

```
EXTEND A ADD exp AS Z
```

puede representarse en SQL como

```
SELECT A.*,  exp' AS Z
FROM   ( A ) AS A
```

La expresión *exp'* en la cláusula SELECT es la contraparte de SQL del operando *exp* de EXTEND. La A entre paréntesis en la cláusula FROM es una <*referencia a tabla*> de complejidad arbitraria (que corres-ponde al operando *A* de EXTEND); la otra *A* en la cláusula FROM es un nombre de variable de alcana Con respecto a SUMMARIZE, el problema básico es que la expresión del álgebra relacional

```
SUMMARIZE A PER B ...
```

---

*Decidimos pasar por alto la idea de que SQL requeriría de hecho de una <*referencia a tabla*> así, **para in-**cluir una definición de variable de alcance sin sentido (vea el apéndice A).

```
A WHERE p A
{x,y,...,z} A
TIMES B A
UNION B A
MINUS B
```

produce un resultado con cardinalidad igual a la de *B*, mientras que el "equivalente" de SQL

```
SELECT ...
FROM   A
GROUP  BY C ;
```

produce un resultado con cardinalidad igual a la de la proyección de *A* sobre *C*.

7.11 SQL no soporta las comparaciones relaciónales en forma directa. Sin embargo, estas opera ciones se pueden simular, aunque sólo de una manera complicada. Por ejemplo, la comparación

```
A = S
```

(donde *A* y *B* son variables de relación) sólo puede ser simulada mediante la expresión de SQL

```
NOT EXISTS ( SELECT * FROM A
             WHERE NOT EXISTS ( SELECT * FROM B
                                WHERE fila-A = fila-B )  )
```

(donde *fila-A y fila-B* son *<constructores de filas>* —vea el apéndice A— que representan una fila completa de A y una fila completa de *B*, respectivamente).

7.12 Aquí tenemos algunas formulaciones de este tipo. Observe que la siguiente lista ni siquiera se acerca a ser completa [4.18]. ¡También observe que ésta es una consulta muy sencilla!

```
SELECT DISTINCT V.PROVEEDOR
FROM    V WHERE   V.V# IN
      ( SELECT VP.V# FROM    VP
        WHERE   VP.P# =  'P2' ) ;

SELECT DISTINCT T.PROVEEDOR FROM
( V NATURAL JOIN VP ) AS T WHERE
T.P# =  'P2' ;

SELECT DISTINCT T.PROVEEDOR
FROM  ( V JOIN VP ON V.V# ■ VP.V# AND VP.P# ■  'P2'  ) AS T  ;

SELECT DISTINCT T.PROVEEDOR FROM
( V JOIN VP USING V# ) AS T WHERE
T.P# =  'P2' ;

SELECT DISTINCT V.PROVEEDOR
FROM    V WHERE   EXISTS
      ( SELECT *
        FROM    VP
        WHERE   VP.V# = V.V#
        AND     VP.P# =  'P2' ) ;

SELECT DISTINCT V.PROVEEDOR
FROM    V, VP
WHERE   V.V# = VP.V#
AND     VP.P# =  'P2' ;

SELECT DISTINCT V.PROVEEDOR
FROM    V WHERE   0 <
      ( SELECT COUNT (*)
        FROM    VP
        WHERE   VP.V# ■ V.V#
        AND     VP.P# =  'P2' ) ;
```

```
SELECT DISTINCT V.PROVEEDOR
FROM  V WHERE  'P2' IN
     ( SELECT VP.P#
       FROM  VP
       WHERE VP.V# = V.V# ) ;

SELECT V.PROVEEDOR FROM
V, VP WHERE V.V# =
VP.V# AND   VP.P# = 'P2'
GROUP BY V.PROVEEDOR ;
```

Pregunta adicional: ¿Cuáles son las implicaciones de lo anterior?

7.13 Numeramos las siguientes soluciones como 7.13.n, en donde *6.n* es el número del ejercicio origi -nal del capítulo 6. Damos por hecho que VX, VY, PX, PY, YX, Y Y, VPYX, VPYY (etcétera) sa variables de alcance que abarcan proveedores, partes, proyectos y envíos, respectivamente; mostramos las definiciones de estas variables de alcance.

7.13.13 `YX`

7.13.14 `YX WHERE YX.CIUDAD = 'Londres'`

7.13.15 `VPYX.V# WHERE VPYX.Y# = Y# ( 'Y1' )`

7.13.16
```
VPYX WHERE VPYX.CANT > CANT ( 300 ) AND
           VPYX.CANT < CANT ( 750 )
```

7.13.17 `( PX.COLOR, PX.CIUDAD )`

7.13.18
```
( VX.V#, PX.P#, YX.Y# ) WHERE VX.CIUDAD ■ PX.CIUDAD
                        AND  PX.CIUDAD = YX.CIUDAD
                        AND  YX.CIUDAD = VX.CIUDAD
```

7.13.19
```
( VX.V#, PX.P#, YX.Y# ) WHERE VX.CIUDAD * PX.CIUDAD
                        OR   PX.CIUDAD * YX.CIUDAD
                        OR   YX.CIUDAD * VX.CIUDAD
```

7.13.20
```
( VX.V#, PX.P#, YX.Y# ) WHERE VX.CIUDAD <> PX.CIUDAD
                        AND  PX. CIUDAD <> YX. CIUDAD
                        AND  YX.CIUDAD * VX.CIUDAD
```

7.13.21
```
VPYX.P# WHERE EXISTS VX ( VX.V# = VPYX.V# AND
                          VX.CIUDAD = 'Londres' )
```

7.13.22
```
VPYX.P# WHERE EXISTS VX EXISTS YX
            ( VX.V# = VPYX.V# AND VX.CIUDAD ■ 'Londres' AND
              YX.Y# = VPYX.Y# AND YX.CIUDAD = 'Londres' )
```

7.13.23
```
( VX.CIUDAD AS CIUDADV, YX.CIUDAD AS CIUDADY )
WHERE EXISTS VPYX ( VPYX.V# = VX.V# AND VPYX.Y# = YX.Y# )
```

7.13.24
```
VPYX.P# WHERE EXISTS VX EXISTS YX
            ( VX.CIUDAD = YX.CIUDAD AND
              VPYX.V#  = VX.V# AND
              VPYX.Y#  = YX.Y# )
```

```
7.13.25 VPYX.Y# WHERE  EXISTS VX  EXISTS YX
                     (  VX.CIUDAD <> YX.CIUDAD AND
                        VPYX.V#     ■ VX.V# AND
                        VPYX.Y#     = YX.Y#  )

7.13.26 ( VPYX.P# AS XP#,  VPYY.P# AS  YP#  )

        WHERE VPYX.V# = VPYY.V# AND VPYX.P# < VPYY.P#*

7.13.27 COUNT  ( VPYX.Y# WHERE  VPYX.V# = V#  (  'V1'  ) ) AS  N

7.13.28  SUM  ( VPYX WHERE  VPYX.V# = V#  (  'V1'  )

                    AND    VPYX.P# =  P#  (  'P1'  ),  CANT  ) AS C
```

Nota: La siguiente "solución" *no* es correcta ¿por qué?:

```
      SUM  ( VPYX.CANT WHERE VPYX.V# = V#  (  'V1'  )
                       AND    VPYX.P# =  P#  (  'P1'  ) ) AS C
```

Respuesta: Debido a que no se pueden eliminar valores duplicados de CANT antes de calcular la suma.

```
7.13.29  ( VPYX.P#, VPYX.Y#,
           SUM ( VPYY WHERE VPYY.P# = VPYX.P#
                       AND    VPYY.Y# = VPYX.Y#, CANT ) AS C )

7.13.30  VPYX.P# WHERE
         AVG ( VPYY WHERE VPYY.P# = VPYX.P#
                     AND VPYY.Y# = VPYX.Y#, CANT ) > CANT ( 350 )

7.13.31  YX. PROYECTO WHERE EXISTS VPYX ( VPYX.Y# = YX.Y# AND
                                          VPYX.V# = V* ( 'V1' ) )

7.13.32 PX.COLOR WHERE EXISTS VPYX ( VPYX.P* = PX.P# AND
                                     VPYX.V# = V# ( 'V1 ' ) )

7.13.33  VPYX.P# WHERE  EXISTS YX  ( YX.CIUDAD =  'Londres'  AND
                                     YX.Y# • VPYX.Y#  )

7.13.34 VPYX.Y# WHERE EXISTS VPYY ( VPYX. P# = VPYY.P# AND
                                    VPYY.V# = V# ( 'V1' ) )

7.13.35 VPYX.V# WHERE EXISTS VPYY EXISTS VPYZ EXISTS PX
                    ( VPYX.P# = VPYY.P# AND VPYY.V#
                      = VPYZ.V# AND VPYZ.P# = PX.P#
                      AND PX.COLOR = COLOR ( 'Rojo'
                      ) )

7.13.36 VX.V# WHERE EXISTS VY ( VY.V# = V# ( 'V1 ' ) AND

                                VX. STATUS < VY. STATUS )

7.13.37 YX.Y# WHERE FORALL YY ( YY.CIUDAD > YX.CIUDAD )
```

O bien, YX.Y# WHERE YX.CIUDAD = MIN (YY.CIUDAD)

```
7.13.38 VPYX.Y# WHERE VPYX.P# = P# ( 'P1' ) AND
              AVG ( VPYY WHERE VPYY.P# = P# ( 'P1' )
                          AND  VPYY.Y# = VPYX.Y#, CANT ) >
              MAX ( VPYZ.CANT WHERE VPYZ.Y# = Y# ( 'Y1' ) )
```

J

```
7.13.39 VPYX.V# WHERE VPYX.P# = P# ( 'P1' )
                AND  VPYX.CANT >
                      AVG ( VPYY
                            WHERE VPYY.P# = P# ( 'P1 ' ) AND
                            VPYY.Y# = VPYX.Y#, CANT

7.13.40 YX.Y# WHERE NOT EXISTS VPYX EXISTS VX EXISTS PX
                  ( VX.CIUDAD - 'Londres' AND
                    PX.COLOR = COLOR ( 'Rojo' ) AND
                    VPYX.V# = VX.V# AND VPYX.P# =
                    PX.P# AND VPYX.Y# • YX.Y# )

7.13.41 YX.Y# WHERE FORALL VPYY ( IF VPYY.Y# = YX.Y#
                                  THEN VPYY.V# = V# ( 'V1'
                                  END IF )

7.13.42 PX.P# WHERE FORALL YX
                  ( IF YX.CIUDAD - 'Londres' THEN
                    EXISTS VPYY ( VPYY.P# - PX.P# AND
                                  VPYY.Y# ■ YX.Y# )
                    END IF )

7.13.4  VX.V# WHERE EXISTS PX        YX EXISTS VPYY
                  ( VPYY.V# = VX  .V  AND
                    VPYY.P# = PX  .P  AND
                    VPYY.Y# = YX  .Y  )

7.13.4  YX.Y# WHERE FORALL VPYY (   VPYY.V# = V# ( 'V1' ) THEN
                                 EXISTS VPYZ
                                 ( VPYZ.Y# = YX.Y# AND
                                 VPYZ.P# - VPYY.P# )
                                 END IF )

7.13.4  RANGEVAR RVX RANGES
        OVER
                   ( VX.CIUDAD ),  ( PX.CIUDAD ), ( YX.CIUDAD
        RVX.CIUDAD

7.13.4  VPYX.P# WHERE EXISTS VX  (   .V# = VPYX.V# AND
                                 VX
                                 V  .CIUDAD = 'Londres' )
               OR   EXISTS YX    (   .Y# = VPYX.Y# AND
                                 Y  .CIUDAD = 'Londres' )

7.13.4  ( VX.V#, PX.P# )
        WHERE NOT EXISTS VPYX (  VPY .V# = VX.V# AND
                                 VPY .P# = PX.P# )

7.13.48 ( VX.V# AS XV#, VY.V# AS YV# )
        WHERE FORALL PZ
          ( ( IF   EXISTS VPYX ( VPYX.V# = VX.V# AND
                                 VPYX.P# = PZ.P# )
               THEN EXISTS VPYY ( VPYY.V# = VY.V# AND
                                 VPYY.P# = PZ.P#
               END IF )
            AND
               IF
                    EXISTS VPYY ( VPYY.V# = VY.V# AND
                                 VPYY.P# = PZ.P# )
               THEN EXISTS VPYX ( VPYX.V# = VX.V# AND
                                 VPYX.P* ■ PZ.P# )
               END IF ) )
```

```
7.13.49 ( VPYX.V#, VPYX.P#, ( VPYY.Y#, VPYY.CANT WHERE
                              VPYY.V# = VPYX.V# AND
                              VPYY.P# = VPYX.P# ) AS YC )
```

7.13.50 Denote R el resultado de evaluar la expresión que mostramos en la solución anterior. Entonces:

```
RANGEVAR RX RANGES OVER R ,
RANGEVAR RY RANGES OVER RX.YC ;

( RX.V#, RX.P#, RY.Y#, RY.CANT )
```

Estamos ampliando ligeramente la sintaxis y la semántica de <*definición de variable de alcance*>. La idea es que la definición de RY dependa de la de RX (observe que las dos definiciones están separadas por una coma, no por un punto y coma; de ahí que puedan agruparse en una sola operación). Para una mayor explicación, vea la referencia [3.3].

7.14 Numeramos las siguientes soluciones como 7.14.n, donde *6.n* es el número del ejercicio original del capítulo 6. Seguimos las mismas convenciones que en la sección 7.6 con respecto a la definición y la denominación de las variables de alcance.

```
7.14.13 ( YX, NOMBREX, CIUDADX )
        WHERE Y ( Y#:YX, PROYECTO:NOMBREX, CIUDAD:CIUDADX )

7.14.14 ( YX, NOMBREX, 'Londres' AS CIUDAD )

        WHERE Y ( Y#:YX, PROYECTO:NOMBREX, CIUDAD: ' Londres' )

7.14.15 VX WHERE VPY ( V#:VX, Y#:Y#('Y1') )

7.14.16 ( VX, PX, YX, CANTX )
        WHERE VPY ( V#:VX, P#:PX, Y#:YX, CANT:CANTX )
        AND  CANTX > CANT ( 300 ) AND CANTX < CANT ( 750 )

7.14.17 ( COLORX, CIUDADX ) WHERE P ( COLOR:COLORX, CIUDAD:CIUDADX )

7.14.18 ( VX, PX, YX ) WHERE EXISTS CIUDADX
                           ( V ( V#:VX, CIUDAD:CIUDADX ) AND
                             P ( P#:PX, CIUDAD:CIUDADX ) AND
                             Y ( Y#:YX, CIUDAD:CIUDADX ) )

7.14.19 ( VX, PX, YX )
        WHERE EXISTS CIUDADX EXISTS CIUDADY EXISTS CIUDADZ (
                       V ( V#:VX, CIUDAD:CIUDADX ) AND P (
                       P#:PX, CIUDAD:CIUDADY ) AND
                         Y ( Y#:YX, CIUDAD:CIUDADZ )
                         AND ( CIUDADX <> CIUDADY OR
                               CIUDADY <> CIUDADZ OR
                               CIUDADZ <> CIUDADX ) )

7.14.20 ( VX, PX, YX )
        WHERE EXISTS CIUDADX EXISTS CIUDADY EXISTS CIUDADZ (
                       V ( V#:VX, CIUDAD:CIUDADX ) AND P (
                       P#:PX, CIUDAD:CIUDADY ) AND
                         Y ( Y#:YX, CIUDAD:CIUDADZ )
                         AND ( CIUDADX <> CIUDADY AND
                               CIUDADY * CIUDADZ AND
                               CIUDADZ <> CIUDADX ) )

7.14.21 PX WHERE EXISTS VX ( VPY ( P#:PX, V#:VX ) AND
                             V ( V#:VX, CIUDAD:'Londres' ) )
```

```
7.14.22 PX WHERE EXISTS VX EXISTS YX
               ( VPY ( V#:VX, P#:PX, Y#:YX ) AND V (
                 V#:VX, CIUDAD:'Londres' ) AND Y (
                 Y#:YX, CIUDAD:'Londres' ) )

7.14.23 ( CIUDADX AS CIUDADV, CIUDADZ AS CIUDADY )
                 WHERE EXISTS VX EXISTS YZ
                      ( V ( V#:VX, CIUDAD:CIUDADX ) AND Y
                        ( Y#:YZ, CIUDAD:CIUDADZ ) AND VPY
                        ( V#:VX, Y#:YZ ) )

7.14.24 PX WHERE EXISTS VX EXISTS YX EXISTS CIUDADX
               ( V ( V#:VX, CIUDAD:CIUDADX ) AND Y
                 ( Y#:YX, CIUDAD:CIUDADX ) AND VPY
                 ( V#:VX, P#:PX, Y#:YX ) )

7.14.25 YY WHERE EXISTS VX EXISTS CIUDADX EXISTS CIUDADY
               ( VPY ( V#:VX, Y#:YY ) AND V (
                 V#:VX, CIUDAD:CIUDADX ) AND Y (
                 Y#:YY, CIUDAD:CIUDADY ) AND
                 CIUDADX <> CIUDADY )

7.14.26 ( PX AS XP#, PY AS YP# ) WHERE EXISTS VX
                                    ( VPY ( V#:VX, P#:PX )
                                      AND VPY ( V#:VX, P#:PY
                                      AND PX < PY  )
```

7.14.27-7.14.30 Omitimos las soluciones.

```
7.14.31 NOMBREX WHERE EXISTS YX
                   ( Y ( Y#:YX, PROYECTO:NOMBREX )
                     AND VPY ( V#:V#('V1'), Y#:YX ) )

7.14.32 COLORX WHERE EXISTS PX
                  ( P ( P#:PX, COLOR:COLORX ) AND
                    VPY ( V#:V#('V1¹), P#:PX ) )

7.14.33 PX WHERE EXISTS YX
               ( VPY ( P#:PX, Y#:YX ) AND
                 Y ( Y#:YX, CIUDAD:'Londres' ) )

7.14.34 YX WHERE EXISTS PX
               (  VPY  (  P#:PX,  Y#:YX  )  AND
                 VPY ( P#:PX, V#:V#('V1') ) )

7.14.35 VX WHERE EXISTS PX EXISTS VY EXISTS PY
               (  VPY  (  V#:VX,  P#:PX  )  AND  VPY  (
                 P#:PX,  V#:VY  )  AND  VPY  (  V#:VY,
                 P#:PY    )    AND    P    (    P#:PY,
                 COLOR:COLOR('Rojo') ) )

7.14.36 VX WHERE EXISTS STATUSX EXISTS STATUSY
               ( V ( V#:VX, STATUS:STATUSX ) AND
                 V  ( V#:V#('Vr), STATUS:STATUSY ) AND
                 STATUSX < STATUSY )
```

```
7.14.37 YX WHERE EXISTS CIUDADX
             ( Y ( Y#:YX, CIUDAD:CIUDADX ) AND FORALL
               CIUDADY ( IF Y ( CIUDAD:CIUDADY ) THEN
               CIUDADY > CIUDADX END IF ) )
```

7.14.38-7.14.39 Omitimos las soluciones.

```
7.14.40 YX WHERE Y ( Y#:YX ) AND
                NOT EXISTS VX EXISTS PX
                  ( VPY ( V#:VX, P#:PX, Y#:YX ) AND V (
                    V#:VX, CIUDAD:'Londres' ) AND P (
                    P#:PX, COLOR:COLORÍ'Rojo') ) )

7.14.41 YX WHERE Y ( Y#:YX )
          AND  FORALL VX ( IF VPY ( V#:VX, Y#:YX )
                           THEN VX • V#('V1') END IF
                           )

7.14.42 PX WHERE P ( P#:PX )
          AND  FORALL YX ( IF Y ( Y#:YX, CIUDAD:'Londres' )
                           THEN VPY ( P#:PX, Y#:YX ) END IF )

7.14.43 VX WHERE V ( V#:VX )
          AND  EXISTS PX FORALL YX
              ( VPY ( V#:VX, P#:PX, Y#:YX ) )

7.14.44 YX WHERE Y ( Y#:YX )
          AND  FORALL PX ( IF VPY ( V#:V#('V1'), P#:PX )
                           THEN VPY ( P#:PX, Y#:YX ) END IF
                           )

7.14.45 CIUDADX WHERE V ( CIUDAD:CIUDADX )
                OR   P ( CIUDAD:CIUDADX )
                OR   Y ( CIUDAD:CIUDADX )

7.14.46 PX WHERE EXISTS VX ( VPY ( V#:VX, P#:PX ) AND
                           V ( V#:VX, CIUDAD:'Londres' ) )
          OR   EXISTS YX ( VPY ( Y#:YX, P#:PX ) AND
                           Y ( Y#:YX, CIUDAD:'Londres' ) )

7.14.47 ( VX, PX ) WHERE V ( V#:VX ) AND P ( P#:PX )
                       AND NOT VPY ( V#:VX, P#:PX )
```

7.14.48 (VX AS XV#, VY AS YV#)

```
       WHERE V ( V#:VX ) AND V ( V#:VY ) AND FORALL PZ
         ( ( IF VPY ( V#:VX, P#:PZ ) THEN VPY ( V#:VY, P#:PZ )
             END IF )
           AND
           ( IF VPY ( V#:VY, P#:PZ ) THEN VPY ( V#:VX, P#:PZ )
             END IF ) )
```

7.14.49-7.14.50 Omitimos las soluciones.

7.15 Numeramos las siguientes soluciones como 7.15.n; donde 6.n es el número del eje: nal en el capítulo 6.

```
7.15.13 SELECT *
        FROM   Y ;
```

O simplemente:

```
      TABLE Y ;
```

```
7.15.14 SELECT Y.*
        FROM  Y
        WHERE Y.CIUDAD = 'Londres' ;
```

```
7.15.15 SELECT DISTINCT VPY.V#
        FROM  VPY
        WHERE VPY.Y# = 'Y1' ;
```

```
7.15.16 SELECT VPY.*
        FROM  VPY
        WHERE VPY.CANT >= 300 AND
        VPY.CANT <= 750 ;
```

```
7.15.17 SELECT DISTINCT P.COLOR, P.CIUDAD
        FROM  P ;
```

```
7.15.18 SELECT V.V#, P.P#, Y.Y#
        FROM  V, P, Y
        WHERE  V.CIUDAD  =  P.CIUDAD
        AND   P.CIUDAD = Y.CIUDAD ;
```

```
7.15.19 SELECT V.V#, P.P#, Y.Y#
        FROM  V, P, Y
        WHERE NOT ( V.CIUDAD = P.CIUDAD AND
                    P.CIUDAD = Y.CIUDAD ) ;
```

```
7.15.20 SELECT V.V#, P.P#, Y.Y#
        FROM  V, P, Y
        WHERE  V.CIUDAD  <>  P.CIUDAD
        AND  P.CIUDAD  <>  Y.CIUDAD
        AND   Y.CIUDAD <> V.CIUDAD ;
```

```
7.15.21 SELECT DISTINCT VPY.P#
        FROM  VPY
        WHERE  ( SELECT V.CIUDAD FROM  V WHERE V.V#
                   = VPY.V# ) ■ 'Londres' ;
```

```
7.15.22 SELECT DISTINCT VPY.P#
        FROM  VPY
        WHERE  ( SELECT V.CIUDAD
                   FROM  V
                   WHERE V.V# = VPY.V# ) = 'Londres'
        AND    ( SELECT Y.CIUDAD
                   FROM  Y
                   WHERE Y.Y# • VPY.Y# ) = 'Londres' ;
```

7.15.23
```
SELECT DISTINCT V.CIUDAD AS CIUDADV,  Y.CIUDAD AS CIUDADY
FROM   V, Y
WHERE  EXISTS
     ( SELECT *
       FROM   VPY
       WHERE  VPY.V# = V.V#
       AND    VPY.Y# = Y.Y# ) ;
```

7.15.24
```
SELECT DISTINCT VPY.P#
FROM   VPY
WHERE   ( SELECT V.CIUDAD
          FROM   V
        WHERE  V.V# = VPY.V# ) = (
        SELECT Y.CIUDAD FROM   Y
        WHERE  Y.Y# = VPY.Y# ) ;
```

7.15.25
```
SELECT DISTINCT VPY.Y#
FROM   VPY
WHERE   ( SELECT V.CIUDAD
          FROM   V
        WHERE  V.V# = VPY.V# ) <> (
        SELECT Y.CIUDAD FROM   Y
        WHERE  Y.Y# = VPY.Y# ) ;
```

7.15.26
```
SELECT DISTINCT VPYX.P0  AS PA,  VPYY.P# AS PB
FROM    VPY AS VPYX, VPY AS VPYY
WHERE  VPYX.V# = VPYY.V#
AND    VPYX.P# < VPYY.P# ;
```

7.15.2
```
SELECT COUNT  ( DISTINCT VPY.Y# ) AS N
FROM   VPY
WHERE  VPY.V  = 'V1' ;
```

7.15.2
```
SELECT SUM (  VPY.CANT ) AS X
FROM   VPY
WHERE  VPY.V  • 'V1'
AND    VPY.P  = 'P1 ' ;
```

7.15.2
```
SELEC  VPY.P  , VPY.Y#, SUM ( VPY.CANT ) AS
T
FROM   VPY
GROUP  BY     ,P#, VPY.Y# ;
```

7.15.3
```
SELEC  DISTINCT VPY.P#
T
FROM   VPY
GROUP  BY     .P#, VPY.Y#
HAVING AVG (  VPY.CANT ) > 350 ;
```

7.15.3
```
SELECT DISTINCT Y.PROYECTO
FROM   Y,
WHERE  Y.Y#   VPY.Y#
AND    VPY.V  - 'V1' ;
```

7.15.3
```
SELECT DISTINCT P.COLOR
FROM   P,
WHERE  P.P#   VPY.P#
AND    VPY.V  = 'V1' ;
```

```
7.15.33 SELECT DISTINCT VPY.P#
        FROM  VPY, Y
        WHERE VPY.Y# = Y.Y#
        AND   Y.CIUDAD ■ 'Londres' ;

7.15.34 SELECT DISTINCT VPYX.Y#
        FROM  VPY AS VPYX, VPY AS VPYY
        WHERE VPYX.P# = VPYY.P# AND
        VPYY.V# = 'V1' ;

7.15.35 SELECT DISTINCT VPYX.V#
        FROM  VPY AS VPYX, VPY AS VPYY, VPY AS VPYZ
        WHERE VPYX.P# = VPYY.P# AND   VPYY.V# =
        VPYZ.V# AND  ( SELECT P.COLOR
               FROM P
               WHERE P.P# = VPYZ.P# ) - 'Rojo' ;

7.15.36 SELECT V.V#
        FROM  V
        WHERE  V.STATUS < ( SELECT V.STATUS FROM V
                            WHERE V.V# = 'V1' ) ;

7.15.37 SELECT Y.Y#
        FROM  Y
        WHERE Y. CIUDAD ■ ( SELECT MIN ( Y.CIUDAD )
                              FROM Y ) ;

7.15.38 SELECT DISTINCT VPYX.Y#
        FROM  VPY AS VPYX
        WHERE VPYX.P# = 'P1 '
        AND  ( SELECT AVG ( VPYY.CANT )
             FROM  VPY AS VPYY WHERE
             VPYY.Y# = VPYX.Y# AND
             VPYY.P# = ' P1 ' ) > (
             SELECT MAX ( VPYZ.CANT )
             FROM  VPY AS VPYZ WHERE
             VPYZ.Y# = ' Y1 ' ) ;

7.15.39 SELECT DISTINCT VPYX.V#
                              FROM  VPY AS VPYX
        WHERE                'P1 ' V
        AND   PYX.P# •
                VPYX.CANT
                              SELECT AVG ( VPYY.CANT )
                              FROM  VPY AS VPYY
                              WHERE VPYY.P# = 'P1'
                              AND   VPYY.Y# - VPYX.Y#

7.15.40 SELECT Y.Y#
FROM  Y
WHERE  NOT EXISTS
    ( SELECT  *
      FROM   VPY, P,   V
      WHERE  VPY.Y# = Y.Y#
      AND    VPY.P# = P.P#
      AND    VPY.V# = V.V#
      AND    P. COLOR  = 'Rojo'
      AND    V.CIUDAD = 'Londres' )
```

```
7.15.41 SELECT Y.Y#
        FROM
        WHERE
               NOT EXISTS
               SELECT *
               FROM  VPY
               WHERE VPY.Y# = Y.Y#
               AND   NOT ( VPY.V# = 'V1'
```

```
7.15.42 SELECT P.P#
        FROM  P
        WHERE NOT EXISTS
              ( SELECT *

                    Y.CIUDAD - 'Londres'
                    NOT EXISTS ( SELECT •
                    FROM  VPY
                      WHERE VPY.P# = P.P#
                      AND   VPY.Y# ■ Y.Y# )

              FROM  Y
              WHERE AND
```

```
7.15.43 SELECT V.V#
        FROM  V
        WHERE EXISTS
        ( SELECT FROM
        WHERE
                     NOT EXISTS
                     SELECT *
                     FROM  Y
                     WHERE NOT EXISTS (
                           SELECT * FROM
                           VPY
                             WHERE VPY.V# = V.V#
                             AND  VPY.P#  =  P.P#
                             AND   VPY.Y# = Y.Y#
```

```
7.15.44 SELECT Y.Y#
FROM  Y
WHERE NOT EXISTS
    ( SELEC  *
      FROM   VPY AS  VPYX
      WHERE  VPYX.V# =  '\
      AND    NOT EXISTS
           ( SELECT  *
             FROM    VPY AS  VPYY
             WHERE   VPYY .  = VPYX.P#
             AND     VPYY .Y = Y.Y# )
```

```
7.15.45 SELECT V.CIUDAD  FROM V
        UNION
        SELECT P.CIUDAD FROM P
        UNION
        SELECT Y.CIUDAD FROM Y ;
```

```
7.15.46 SELECT DISTINCT VPY.P#
        FROM   VPY
        WHERE  ( SELECT V.CIUDAD
                  FROM   V
                  WHERE V.V# ■ VPY.V# ) ■ 'Londres'
        OR     ( SELECT Y.CIUDAD
                  FROM   Y
                  WHERE Y.Y# = VPY.Y# ) ■ 'Londres'
```

7.15.47
```
SELECT V.V#, P.P#
       FROM  V, P
       EXCEPT
       SELECT VPY.V#, VPY.P#
       FROM  VPY ;
```

7.15.48 Omitimos la solución.

7.15.49-7.15.50 No pueden hacerse.

CAPITULO 8

Integridad

8.1 INTRODUCCIÓN

El término **integridad** se refiere a la *exactitud* o *corrección* de los datos en la base de datos. Como señalamos en el capítulo 3, una base de datos determinada podría estar sujeta a cualquier cantidad de restricciones de integridad (en general) de una complejidad arbitraria. Por ejemplo, en el caso de la base de datos de proveedores y partes, es probable que los números de proveedor tuvieran que ser de la forma *Vnnnn (nnnn* - hasta cuatro dígitos decimales) y ser únicos; que los valores de status tuvieran que estar en el rango de 1 a 100; que los proveedores de Londres tuvieran un status de 20; que las cantidades de envío tuvieran que ser múltiplos de 50; que las partes rojas tuvieran que estar almacenadas en Londres; y así sucesivamente. Por lo tanto, comúnmente el DBMS necesita estar informado de dichas restricciones y por supuesto, necesita hacerlas cumplir de alguna manera (básicamente rechazando cualquier actualización que de otro modo las violaría). Por ejemplo (una vez más en **Tutorial D):**

```
CONSTRAINT R3V
    IS_EMPTY ( V WHERE STATUS  < 1  OR STATUS > 100 )
```

("los valores de status deben estar en el rango de 1 a 100"). Observe el *nombre* de restricción R3V ("restricción tres de proveedores"); la restricción será registrada en el catálogo del sistema bajo ese nombre, y dicho nombre aparecerá en cualquier mensaje de diagnóstico que el sistema emita en respuesta a un intento de violar la restricción. La propia restricción está especificada como una expresión lógica que no debe dar como resultado *falso.*

Nota: Tomamos la versión algebraica de **Tutorial D** por concluyente; como consecuencia, la expresión lógica tomará a menudo, aunque no invariablemente, la forma IS_EMPTY (...), que significa que no hay información en la base de datos que viole la restricción en cuestión (vea el capítulo 6, sección 6.9). Una equivalencia en el cálculo del ejemplo que acabamos de mostrar podría lucir como la siguiente:*

```
CONSTRAINT R3V
    FORALL VX  ( VX.STATUS  > 1  AND VX.STATUS < 100 ) ;
```

(donde VX es por supuesto una variable de alcance que abarca a los proveedores).

*En la práctica, con frecuencia parece ser más fácil formular las restricciones (en especial las complicadas) en términos del cálculo en lugar del álgebra. En este capítulo nos concentramos en el álgebra para ser consistentes con las explicaciones presentadas en otras partes del libro, pero tal vez a usted le gustaría intentar el ejercicio de convertir algunos de los ejemplos que siguen a la forma del cálculo.

Al margen, subrayamos que la expresión lógica en una restricción del cálculo debe ser una *WFF cerrada* (vea el capítulo 7, sección7.2) y con frecuencia, aunque no invariablemente, tomará la forma FORALL *x* (...). Por lo tanto, observe que el ejemplo especifica que *todos* los valores de status de los proveedores deben estar en el rango indicado. Por supuesto, en la práctica basta con que el sistema verifique sólo al proveedor recién insertado o actualizado, no a todos los proveedores.

Al declarar una nueva restricción, el sistema primero debe asegurarse de que la base de datos la satisfaga actualmente. Si no es así, la nueva restricción es rechazada; en caso contrario, es aceptada (es decir, se guarda en el catálogo) e impuesta desde ese momento. En el ejemplo que nos ocupa, la imposición de la restricción requiere que el DBMS supervise todas las operaciones que podrían insertar un nuevo proveedor o modificar el status de un proveedor ya existente.

Desde luego, también necesitamos una forma de deshacernos de restricciones ya existentes:

```
DROP CONSTRAINT <nombre de restricción> ;
```

Por ejemplo:

```
DROP CONSTRAINT R3V ;
```

Nota: Como indica la explicación anterior, estamos interesados de manera muy específica en el soporte de la integridad **declarativa.** Lamentablemente, pocos productos actuales ofrecen mucho en materia de este soporte. Aunque la situación está mejorando lentamente al respecto, aún se da el caso de que algunos productos (en especial los no relaciónales) enfatizan de manera muy específica el enfoque opuesto; es decir, el soporte *mediante procedimientos,* utilizando **procedimientos almacenados** o **activados.*** Pero se ha sugerido que si el DBMS proporcionara el soporte declarativo, entonces hasta un 90 por ciento de la definición de una base de datos típica consistiría en restricciones; por lo tanto, un sistema que proporcionara dicho soporte liberaría a los programadores de aplicaciones de una carga considerable y les permitiría hacerse significativamente más productivos. El soporte de la integridad declarativa es importante.

Antes de continuar, debemos decir que la parte de integridad del modelo relacional es laque más ha cambiado a través de los años (tal vez deberíamos decir *evolucionado* en lugar de *cambiado).* Como mencionamos en el capítulo 3, en un principio el énfasis estaba puesto específicamente en las claves primaria y externa ("claves" para abreviar). Sin embargo, la importancia —en realidad la importancia **crucial**— de las restricciones de integridad en general, comenzó gradualmente a entenderse mejor y a apreciarse con mayor amplitud; al mismo tiempo comenzaron a surgir ciertas cuestiones difíciles con respecto a las claves. La estructura de este capítulo refleja este cambio en el énfasis, al grado de que primero trata las restricciones de integridad en general (con cierta amplitud) y luego explica las claves, las cuales siguen teniendo gran importancia pragmática.

---

*Los procedimientos almacenados y activados, son procedimientos precompilados que pueden ser llamados desde programas de aplicación. Ejemplos de ellos podrían incluir los operadores definidos por el usuario ABS, DIST, REFLEJO, etcétera, explicados en la sección 5.2 (subseceión "Definición de operadores"). Dichos procedimientos pueden ser considerados de manera lógica como una extensión al DBMS (en un sistema cliente-servidor, a menudo se mantendrán y ejecutarán en el sitio del servidor). En la sección 8.8, tenemos algo más que decir con respecto a dichos procedimientos, así como en los comentarios de algunas referencias que se encuentran al final de este capítulo y del capítulo 20.

Un esquema de clasificación de restricciones

De acuerdo con la referencia [3.3], clasificamos las restricciones de integridad en general en cuatro grandes categorías: restricciones de tipo (dominio), de atributo, de varrel y de base de datos. En esencia:

- Una restricción de *tipo* especifica los valores válidos para un tipo dado. *Nota:* a lo largo de este capítulo, empleamos "tipo" para referirnos específicamente a un tipo *escalar*. Por supuesto, los tipos de relación también están sujetos a las restricciones de tipo, pero dichas restricciones son básicamente sólo una consecuencia lógica de las restricciones de tipo que se aplican a los tipos escalares en términos de los cuales esos tipos de relación están (en úl tima instancia) definidos.
- Una restricción de *atributo* especifica el valor válido de un atributo dado.
- Una restricción de *varrel* especifica los valores válidos de una varrel determinada.
- Una restricción de *base de datos* especifica el valor válido de una base de datos dada.

Los cuatro casos son explicados en detalle en las secciones 8.2 a 8.5, respectivamente.

8.2 RESTRICCIONES DE TIPO

En esencia, una restricción de tipo es (o es equivalente de manera lógica a) una sola enumeración de los valores válidos del tipo. Aquí tenemos un ejemplo sencillo, la restricción del tipo PESO:

```
TYPE PESO POSSREP ( RATIONAL )
     CONSTRAINT THE_PES0 ( PESO ) > 0.0 ;
```

Adoptamos una convención obvia por medio de la cual es posible que una restricción de tipo haga uso del *nombre de tipo* aplicable para denotar un valor cualquiera del tipo en cuestión; por lo tanto, este ejemplo limita los pesos de tal forma que puedan ser representados mediante un número racional que sea mayor que cero. Cualquier expresión que deba dar como resultado un peso pero que de hecho no produzca un valor que satisfaga esta restricción, fracasará. *Nota:* Si necesita refrescar su memoria con respecto a las especificaciones POSSREP y a los operadores THE_ consulte el capítulo 5.

Para repetir, una restricción de tipo es básicamente sólo una especificación de los valores que conforman el tipo en cuestión. Por lo tanto, en **Tutorial D** agrupamos dichas restricciones con la definición del tipo aplicable, y las identificamos por medio del nombre del tipo aplicable. (De aquí que solamente se pueda eliminar una restricción de tipo eliminando el propio tipo.)

Ahora bien, debe quedar claro que en última instancia, la única forma en que *cualquier* expresión pueda producir un valor de tipo PESO es mediante alguna invocación al selector PESO. Por lo tanto, la única forma en que dicha expresión puede violar la restricción del tipo PESO es cuando la invocación al selector en cuestión la viola. Por lo tanto, *siempre podemos pensar, por lo menos conceptualmente, que las restricciones de tipo son verificadas durante la ejecución de alguna invocación al selector.* Como consecuencia, podemos decir que las restricciones de tipo son verificadas *de inmediato* y por lo tanto, que ninguna varrel puede adquirir nunca un valor para ningún atributo de cualquier tupia si éste no es del tipo apropiado (por supuesto, en un sistema que maneja las restricciones de tipo).

Aquí tenemos otro ejemplo de una restricción de tipo:

```
TYPE PUNTO POSSREP CARTESIANO ( X RATIONAL, Y RATIONAL )
     CONSTRAINT ABS ( THE_X ( PUNTO ) )< 100.0 AND ABS
     ( THEY ( PUNTO ) ) £100.0 ;
```

Aquí la verificación de tipo se hace de manera conceptual durante la ejecución de cación al selector CARTESIANO. Observe el uso del operador ABS definido por el usua el capítulo 5, sección 5.2).

Aquí tenemos un tercer ejemplo:

```
TYPE ELIPSE POSSREP ( A LONGITUD, B LONGITUD, CTRO PUNTO )
    CONSTRAINT THE_A ( ELIPSE ) > THE_B ( ELIPSE ) ;
```

Aquí, los componentes de representación posible —es decir A, B y CTRO— la longitud del semieje mayor *a,* la longitud del semieje menor *b* y el punto central erro, pectivamente, de la elipse en cuestión. Suponga que se declara la variable escalar E com ELIPSE y que su valor actual tiene un semieje mayor de longitud cinco y un semieje me longitud cuatro. Ahora, considere la asignación:

```
THE_B ( E ) := LONGITUD ( 6.0 ) ;
```

Nota: Por razones de simplicidad, basamos específicamente este ejemplo en una variableí y en una asignación escalar, pero en vez de ello también lo podríamos haber basado en una\ ble *de relación* (varrel) y en una asignación relacional.

Es claro que la asignación tal como se muestra, fracasará; pero no es la asignado *misma* la que tiene el error. En vez de ello, el error ocurre una vez más dentro de una **invoca** al selector (aunque tal invocación no sea visible directamente en la asignación), ya que la nación que se muestra es en realidad una forma abreviada de la siguiente:*

```
E := ELIPSE ( THE_A ( E ), LONGITUD ( 6.0 ), THE_CTRO ( E ) );
```

Y en este caso, es la invocación de la parte derecha la que falla.

8.3 RESTRICCIONES DE ATRIBUTO

Una restricción de atributo es básicamente sólo una declaración para que un atributo < cado sea de un tipo en particular. Por ejemplo, considere una vez más la definición de de proveedores:

```
VAR V BASE RELATION { V#
    V#, PROVEEDOR NOMBRE,
    STATUS    INTEGER,
    CIUDAD    CHAR } . .
```

*En otras palabras, y a pesar del hecho de que no mencionamos la idea de manera explícita en el ca 5, ¡las seudovariables THE_ son lógicamente innecesarias! Es decir, cualquier asignación ., riable THE_ siempre es equivalente de manera lógica a (y de hecho está definida para ser una fornu viada de) la asignación del resultado de una cierta invocación al selector para una variable norma

En esta varrel, los valores de los atributos V#, PROVEEDOR, STATUS y CIUDAD están restringidos a los tipos V#, NOMBRE, INTEGER y CHAR, respectivamente. En otras palabras, las restricciones de atributo son parte de la definición del atributo en cuestión y pueden ser identificadas por medio del nombre de atributo correspondiente. De aquí que una restricción de atributo sólo pueda ser eliminada mediante la eliminación del propio atributo (lo cual, en la práctica significa generalmente eliminar la varrel que lo contiene).

Nota: En principio, cualquier intento de introducir un valor de atributo que no sea un valor del tipo relevante dentro de la base de datos, será simplemente rechazado. Sin embargo, en la práctica esta situación nunca deberá presentarse en tanto el sistema haga cumplir las restricciones de tipo descritas en la sección anterior.

8.4 RESTRICCIONES DE VARREL

Una restricción de varrel es la que es impuesta a una varrel individual (ésta se expresa solamente en términos de la varrel en cuestión, aunque en otros aspectos puede ser compleja). Aquí tenemos algunos ejemplos:

```
CONSTRAINT R5V
    ISEMPTY ( V WHERE CIUDAD = 'Londres' AND STATUS * 20 ) ;
```

("los proveedores en Londres deben tener un status de 20").

```
CONSTRAINT R4P
    IS_EMPTY ( P WHERE COLOR - COLOR ( 'Rojo' )
                  AND CIUDAD * 'Londres' ) ;
```

("las partes rojas deben almacenarse en Londres").

```
CONSTRAINT RKV
    COUNT ( V ) = COUNT ( V { V# } ) ;
```

("los números de proveedor son únicos"; o de manera más formal, "{V#} es una clave candidata de proveedores". Vea la sección 8.8).

```
CONSTRAINT R7P
    IF NOT ( IS_EMPTY ( P ) ) THEN
       COUNT ( P WHERE COLOR - COLOR ( 'Rojo' ) ) > 0
    END IF;
```

("si existen partes, por lo menos una de ellas debe ser roja"). Por cierto, observe que este ejemplo difiere de todos los que hemos visto, en tanto que las operaciones DELETE también tienen el potencial de violar la restricción.

Las restricciones de varrels siempre son verificadas de inmediato (en realidad, como parte de la ejecución de cualquier instrucción que pudiera ocasionar que fueran violadas). Por lo tanto, cualquier instrucción que intente asignar un valor a una varrel dada que viole cualquier restricción para esa varrel, será en efecto rechazada.

8.5 RESTRICCIONES DE BASE DE DATOS

Una restricción de base de datos es aquella que relaciona dos o más varrels distintas
mos algunos ejemplos:

```
CONSTRAINT R1BD
    IS_EMPTY ( ( V JOIN VP )
               WHERE STATUS < 20 AND CANT > CANT ( 500 ) ) ;
```

("ningún proveedor con status menor que 20 puede suministrar parte alguna en unacanti
superior a 500"). *Ejercicio:* ¿Cuáles son las operaciones de actualización que el DBMS d
monitorear para hacer cumplir la restricción R1BD?

```
CONSTRAINT R2BD VP { V# } < V { V# } ;
```

("todo número de proveedor en la varrel de envíos existe también en la varrel de proveído
recuerde del capítulo 6 que usamos "<" para indicar "subconjunto de"). Debido a que el
buto V# de la varrel V constituye una clave candidata para los proveedores, esta resta
básicamente la restricción *referencial* necesaria de envíos a proveedores (es decir, (V#)i
varrel VP es una clave externa para los envíos que se refieren a proveedores). Para una i
explicación, consulte la sección 8.8.

```
CONSTRAINT R3BD VP { P# } = P { P# } ;
```

("toda parte debe tener por lo menos un envío"). *Nota:* Por supuesto, también es el casoquet
envío debe tener exactamente una parte, en virtud de que {P#} en la varrel P es una claven!
data para partes y existe una restricción referencial de envíos a partes; aquí, no nos ocupnri
mostrar esta última restricción. Una vez más, para una mayor explicación consulte la sección!

Estos dos últimos ejemplos sirven también para ilustrar la idea de que (en general) I;
ficación de restricciones de base de datos no puede hacerse de inmediato, sino que debe *difer*
hasta el final de la transacción; es decir, al momento del COMMIT (consulte el capítulo3
necesita refrescar su memoria con respecto a COMMIT). Suponga el caso contrario, que la \
ficación fuese inmediata y que actualmente no hay ninguna parte ni envío. Entonces, la i
ción de una parte fracasará, ya que viola la restricción R3BD; en forma similar, la inserciónі
un envío fracasará, ya que viola la restricción R2BD.*

Si se viola una restricción de base de datos al momento del COMMIT, la transacción
deshecha.

8.6 LA REGLA DE ORO

*Nota: Advertencia para el lector. El material de esta sección es de importancia funda
Sin embargo, por desgracia, no es soportado ampliamente en la práctica y ni siquiera
entendido, aunque en principio es bastante directo.*

---

*De hecho, la referencia [3.3] propone una forma *múltiple* de asignación que permitiría que partes y en
fueran insertados en una sola operación. Si dichas asignaciones fueran soportadas, las restricciones debí
de datos podrían verificarse de inmediato.

En el capítulo 3, sección 3.4, explicamos cómo cualquier relación dada tenía asociado un *predicado* y cómo las tupias de esa relación denotaban *proposiciones verdaderas* derivadas de ese predicado. Y en el capítulo 5, sección 5.3, mencionamos la *Suposición del mundo cerrado,* la cual dice en efecto que si una cierta tupia no aparece en una cierta relación, entonces podemos dar por hecho que la proposición correspondiente *es falsa.*

Ahora bien, antes no enfatizamos este punto, pero debe quedar claro que una *varrel* también tiene un predicado; es decir, el predicado que es común a todas las relaciones posibles que son valores válidos de la varrel en cuestión. Por ejemplo, considere la varrel de proveedores V. El predicado para esa varrel es similar al siguiente:

> *El proveedor con el número de proveedor especificado (V#), tiene el nombre especificado (PROVEEDOR) y el valor de status especificado (STATUS), y se ubica en la ciudad especificada (CIUDAD); además, dos proveedores no pueden tener el mismo número al mismo tiempo.*

Este enunciado no es ni preciso ni completo, pero servirá para los fines actuales.

También debe quedar claro que en esencia, el predicado sirve como **criterio de aceptabilidad para las actualizaciones** sobre la varrel en cuestión; dicta si una operación INSERT, UPDATE o DELETE en particular puede tener éxito. Por ejemplo, un intento de insertar un nuevo proveedor con un número de proveedor igual al de otro proveedor ya existente, deberá seguramente ser rechazado.

Por lo tanto, de manera ideal el DBMS conocería y entendería el predicado de cada varrel, de modo que pudiera manejar correctamente todos los intentos posibles por actualizar la base de datos. Pero por supuesto, esta meta es inalcanzable. Por ejemplo, no hay forma de que el DBMS pueda saber lo que significa que un cierto proveedor esté "en" una determinada ciudad. Y no hay forma de que pueda saber *a priori* que el predicado de proveedores es tal que (por ejemplo) la tupia

```
V#        : V# ( 'V1'
PROVEEDOR : NOMBRE (   'Smith' )
STATUS    : 20
CIUDAD    : 'Londres'
```

lo satisface, mientras que la tupia

```
{ V#        : V# ( 'V6' )
  PROVEEDOR : NOMBRE ( 'Smith' ) ,
  STATUS    : 50
  CIUDAD    : 'Roma'              }
```

no lo satisface. De hecho, si el usuario final presenta esta tupia para su inserción, todo lo que el sistema puede hacer es asegurar que no viole ninguna restricción de integridad conocida. Si no lo hace, el sistema aceptará entonces la tupia para su inserción *y a partir de ese punto la tratará como una proposición verdadera.*

Por lo tanto, para repetir: el sistema no conoce ni entiende al 100 por ciento (ni puede hacerlo) el predicado de proveedores. *Pero sí conoce una buena aproximación al predicado;* para ser específicos, conoce las restricciones de integridad que se aplican a proveedores. Por lo tanto, *definimos* el **predicado de varrel** para la varrel de proveedores (o de manera más general, de cualquier varrel) como el AND lógico de todas las restricciones de varrel que se aplican a ella. Por ejemplo, el predicado de varrel para la varrel V es parecido al siguiente:

wr


```
( IS_EMPTY ( V WHERE STATUS < 1 OR STATUS > 100 ) ) AND
( IS_EMPTY ( V WHERE CIUDAD = 'Londres' AND STATUS * 20 ) ) AND
( COUNT ( V ) = COUNT ( V { V# } ) )
```

(Además, es claro que el sistema sabe que los atributos V#, PROVEEDOR, STATUS DAD son de los tipos V#, NOMBRE, INTEGER y CHAR, respectivamente.)

Por lo tanto, observe que en efecto existen dos predicados asociados con toda val el predicado informal o **externo** que entienden los usuarios pero no el sistema, y el p formal o **interno** que entienden tanto los usuarios *como* el sistema. Y por supuesto, eselp cado interno al que nos referimos con el término "predicado de varrel" y es este mismoi verificará el sistema siempre que haya un intento de actualizar la varrel en cuestión. **Deh** a partir de este momento tomaremos el término *predicado* (cuando se use en conexiona guna varrel) para referirnos específicamente al predicado interno de esa varrel, salvo i cación explícita en caso contrario.

Dadas las definiciones anteriores, ahora podemos enunciar *La regla de oro* (por 1 la primera versión de ella):

> *Nunca debe permitirse una operación de actualización que deje a cualquier varrel en* w *estado que viole su propio predicado.*

También debemos enfatizar la idea de que aquí "varrel" no significa necesariamenl varrel base; *La regla de oro* se aplica a *todas* las varrels, lo mismo derivadas que base.l remos esta idea en el siguiente capítulo.

Cerramos esta sección señalando que así como toda varrel tiene un predicado asociad bien cada base de datos tiene un predicado asociado: el **predicado de** base de datos p base de datos, el cual definimos como el AND lógico de todas las restricciones de baseded y de varrel que se apliquen a la base de datos en cuestión. De modo que podemos e ampliar *La regla de oro:**

> *Nunca debe permitirse una operación de actualización que deje a cualquier vamlt un estado que viole su propio predicado. En forma similar, nunca debe permitirse* **i** *transacción de actualización que deje a la base de datos en un estado que viole sup predicado.*

8.7 RESTRICCIONES DE ESTADO FRENTE A RESTRICCIONES DE TRANSICIÓN

Hasta ahora, todas las restricciones que hemos expuesto en este capítulo han sido r de **estado,** las cuales se ocuparon de los *estados* correctos de la base de datos. Sin en ocasiones es necesario considerar también restricciones de **transición;** es decir, i sobre transiciones válidas de un estado correcto a otro. Por ejemplo, en una base de c hiciera referencia a personas, podría haber una serie de restricciones de transición queC que ver con cambios en el estado civil. Por ejemplo, las siguientes transiciones son val

---

* Si las asignaciones múltiples fueran soportadas (una posibilidad señalada previamente en una notailp tal vez podríamos dejar *La regla de oro* en su forma original que es más sencilla.

- Soltero a casado
- Casado a viudo
- Casado a divorciado
- Viudo a casado

(etcétera), en tanto que las siguientes no lo son:

- Soltero a viudo
- Soltero a divorciado
- Viudo a divorciado
- Divorciado a viudo

(etcétera). Para retomar la base de datos de proveedores y partes, aquí tenemos otro ejemplo ("nunca debe disminuir el status de proveedor"):

```
CONSTRAINT R1TR IS_EMPTY
   ( ( ( V { V#, STATUS } RENAME STATUS AS STATUS' )
       JOIN V { V#, STATUS } )
     WHERE STATUS' > STATUS ) ;
```

Explicación: Presentamos la convención de que un nombre de varrel con apóstrofo (como V en el ejemplo) se refiere a la varrel correspondiente como estaba *antes de la actualización en consideración.* Por lo tanto, la restricción del ejemplo puede entenderse como sigue: Si (a) juntamos (sobre los números de proveedor) la relación que es el valor de la varrel V antes de la actualización y la relación que es el valor después de ésta, y (b) escogemos de esa junta las tupias para las cuales el valor anterior del status es mayor que el nuevo, entonces (c) el resultado final debe estar vacío. (Puesto que la junta se hace sobre los números de proveedor, toda tupia en el resultado de la junta para la cual el valor anterior del status fuera mayor que el nuevo, representaría un proveedor cuyo status habría disminuido.)

Nota: La restricción R1TR es una restricción de transición de *varrel* (se aplica sólo a una varrel; es decir, a los proveedores) y por lo tanto la verificación es inmediata. En contraste, aquí tenemos un ejemplo de una restricción de transición de *base de datos* ("la cantidad total de cualquier parte dada, considerando a todos los proveedores, nunca puede disminuir"):

```
CONSTRAINT R2TR IS_EMPTY
    ( ( ( SUMMARIZE VP' PER V { V# } ADD SUM ( CANT ) AS VC' )
        JOIN
        ( SUMMARIZE VP PER V { V# } ADD SUM ( CANT ) AS VC ) )
      WHERE VC > VC ) ;
```

La restricción R2TR es una restricción de transición de base de datos (comprende dos varrels distintas: proveedores y envíos); por lo tanto, la verificación es diferida al momento del COMMIT, y los nombres de varrels V y VP' son tomados como representaciones de las varrels V y VP tal como estaban al inicio de la transacción (BEGIN TRANSACTION).

El concepto de las restricciones de estado *frente* a las de transición no tiene sentido para las restricciones de tipo o de atributo.

8.8 CLAVES

El modelo relacional siempre ha puesto énfasis en el concepto de las claves; aunque como hemos visto, en realidad sólo son un caso especial (muy importante) de un fenómeno más general. En esta sección, centraremos nuestra atención específicamente en las claves.

Nota: Aunque aquí las ideas básicas son bastante sencillas, por desgracia existe un factor importante de complicación: los valores *nulos.* La posibilidad de que (por ejemplo) una clave externa determinada pudiera permitir nulos, empaña la imagen de manera considerable. Sin embargo, los nulos constituyen por sí mismos un tema extenso, que sería inadecuado abordar en detalle en este momento. Por lo tanto, por razones pedagógicas, en esta sección ignoraremos casi en su totalidad a los nulos; en el capítulo 18, cuando tratemos los nulos en general, volveremos a explicar el impacto de los nulos sobre las claves. (De hecho, nosotros creemos firmemente que los nulos son un error y que nunca debieron ser presentados, aunque estaría mal ignorarlos por completo en un libro de esta naturaleza.)

Claves **candidatas**

Sea *R* una varrel. Por definición, el conjunto de todos los atributos de *R* tiene la propiedad de **unicidad,** lo que significa que ningún par de tupias que estén dentro del valor de *R* en un momento dado, pueden estar duplicadas entre sí. En la práctica, a menudo se da el caso de que algún subconjunto propio del conjunto de atributos de *R* tiene también la propiedad de unicidad; por ejemplo, en el caso de la varrel de proveedores, *V,* el subconjunto que contiene únicamente al atributo V# tiene esa propiedad. Estos hechos constituyen la intuición que está detrás de la definición de la *clave candidata:*

- Sea *K* un conjunto de atributos de la varrel *R.* Entonces *K* es una **clave candidata** de *R* si, y solamente si, posee las dos propiedades siguientes:*
  a. **Unicidad:** Jamás, ningún valor válido de *R* contiene dos tupias distintas con el mismo valor de *K.*
  b. **Irreductibilidad:** Ningún subconjunto propio de *K* tiene la propiedad de unicidad.

Observe que toda varrel tiene por lo menos una clave candidata. La propiedad de unicidad de dichas claves es muy clara. En cuanto a la propiedad de irreductibilidad, la idea es que si especificáramos una "clave candidata" que *no* fuera irreducible, el sistema no estaría al tanto del estado de las cosas; y por lo tanto, no podría hacer cumplir adecuadamente la restricción de integridad asociada. Por ejemplo, suponga que definiéramos la combinación {V#,CIUDAD} —en vez de sólo {V#}— como la clave candidata para proveedores. Entonces el sistema no haría cumplir la restricción de que los números de proveedor son "globalmente" únicos; en su lugar, sólo haría cumplir la restricción más débil de que los números de proveedor son "localmente" únicos dentro de una ciudad. Por esta razón, entre otras, requerimos que las claves candidatas no incluyan atributo alguno que sea irrelevante para los fines de identificación única^

---

* Observe que la definición se aplica específicamente a las *variables de relación* (varrels). Podemos también definir una noción análoga para los *valores* de relación [3.3], pero el caso importante son las varrels. ᵗOtra buena razón por la que requerimos que las claves candidatas sean irreducibles tiene que ver con la coincidencia con las claves *externas.* Toda clave extema que hiciera referencia (si esto fuese posible) a una clave candidata "reductible", también sería "reductible"; y la varrel que la contuviera estaría seguramente violando los principios de normalización adicional (vea el capítulo 11).

Por cierto, en gran parte de la literatura (incluyendo ediciones anteriores de este libro) a la irreductibilidad (en el sentido anterior) se le conoce como *minimalidad.* Sin embargo, "minimalidad" en realidad no es el *término justo,* ya que decir que la clave candidata *Kl* es "mínima" no significa que no pueda encontrarse otra clave candidata *K2* que tenga menos componentes; es totalmente posible que (por ejemplo) *Kl* tenga cuatro componentes y *K2* sólo dos. Nosotros nos quedaremos con el término "irreductible."

En **Tutorial D,** utilizamos la sintaxis

```
KEY { <lista de nombres de atributo separados con comas> }
```

dentro de una varrel, para especificar una clave candidata de la varrel en cuestión. Aquí tenemos algunos ejemplos:

-
```
VAR V BASE RELATION
    { V#         V#,
      PROVEEDOR NOMBRE,
      STATUS   INTEGER,
      CIUDAD   CHAR }
    KEY { V# } ;
```

  Nota: En capítulos anteriores mostramos esta definición con una cláusula **PRIMARY** KEY, **no** con una cláusula KEY. Para una explicación más amplia, consulte la subsección "Claves primarias y claves alternas" que se encuentra adelante.

-
```
VAR VP BASE RELATION
     {    V #    V # ,
       P #       P # ,
       CANT     CANT }
       KEY { V#, P# } ... ;
```

  Este ejemplo muestra una varrel con una clave candidata **compuesta** (es decir, una clave candidata que involucra más de un atributo). Una clave candidata **simple** es aquella que no es compuesta.

```
VAR ELEMENTO BASE RELATION { NOMBRE       NOMBRE,
                             SÍMBOLO       CHAR,
                             NUMATOMICO  INTEGER }
                       KEY { NOMBRE }
                       KEY { SÍMBOLO }
                       KEY { NUMATOMICO     } ;
```

  Este ejemplo muestra una varrel con varias claves candidatas (simples) distintas.

```
VAR MATRIMONIO BASE RELATION { ESPOSO                NOMBRE,
                               ESPOSA                NOMBRE, */ de matrimonio
/* se da por hecho que no hay  FECHA /*              DATE  }
poliandria, poligamia y que ningún */ /* esposo y esposa se casan
entre si más de una vez ...       */
              KEY { ESPOSO, FECHA }
              KEY { FECHA, ESPOSA }
              KEY { ESPOSA, ESPOSO } ;
```

  Este ejemplo muestra una varrel con varias claves candidatas compuestas distintas (y que se traslapan).

Por supuesto, como señalamos en la sección 8.4, una definición de clave candidata es en realidad una forma abreviada de una cierta restricción de varrel. La forma abreviada es útil, ya

que el concepto de clave candidata es muy importante desde un punto de vista pragmático. Para ser más específicos, las claves candidatas ofrecen el **mecanismo de direccionamiento en** el **nivel de tupia** básico dentro del modelo relacional, lo que significa que la única forma garantizada por el sistema para señalar una tupia específica es por medio de algún valor de clave candidata. Por ejemplo, está garantizado que la expresión

```
V WHERE V# = V# ( 'V3' )
```

produce como máximo una tupia (para ser más precisos, produce una *relación* que contiene como máximo una tupia). En contraste, la expresión

```
V WHERE CIUDAD = 'París'
```

produce en general, un número impredecible de tupias. De aquí que *las claves candidatas sean tan importantes para la operación exitosa de un sistema relacional como lo son las direcciones de memoria principal para la operación exitosa de la máquina subyacente.* En consecuencia:

1. En ciertas circunstancias, las "varrels" que no tienen una clave candidata (es decir, aque llas que permiten tupias duplicadas) están expuestas a mostrar un comportamiento extraño y anormal.
2. En ocasiones, un sistema que no tiene conocimiento de las claves candidatas está expuesto a mostrar un comportamiento que no es "verdaderamente relacional", incluso si las varrels con las que trata son en realidad verdaderas varrels y no permiten tupias duplicadas.

El comportamiento arriba referido como "extraño y anormal" y como "no verdaderamente relacional" tiene que ver con aspectos como la *actualización de vistas* y la *optimization* (vea los capítulos 9 y 17, respectivamente).

Las que siguen son algunas ideas finales para cerrar esta subsección:

- Un superconjunto de una clave candidata es una **superclave.** Por ejemplo, el conjunto de atributos {V#,CIUDAD} es una superclave de la varrel V. Una superclave tiene la pro piedad de unicidad pero no necesariamente la propiedad de irreductibilidad (desde luego, una clave candidata es un caso especial de una superclave).
- Si *SK* es una superclave de la varrel *R,* y A es un atributo de *R,* entonces la **dependencia funcional** *SK* —> A es necesariamente verdadera en *R* (este concepto importante lo explico **a** profundidad en el capítulo 10). De hecho, podemos *definir* una superclave como un subconjunto *SK* de los atributos de *R,* tales que la dependencia funcional *SK* —»A es verdadera para todos los atributos A de *R.*
- Por último, observe que la noción lógica de una clave candidata no debe confundirse con la noción física de un "índice único" (aunque a menudo se emplea este último para implementar la primera). En otras palabras, no hay implicación alguna de que deba existir un índice (o lo que es lo mismo, cualquier otra ruta especial de acceso físico) sobre una clave candidata. En la práctica, probablemente *existirán* algunas de estas rutas especiales de acceso físico, pero el hecho de que existan o no, rebasa el alcance del modelo relacional como tal.

Claves primarias y claves alternas

Como hemos visto, es posible que una varrel determinada tenga más de una clave candidata. En tal caso, el modelo relacional ha requerido de manera histórica —por lo menos en el caso de las varrels *base*— que sólo una de esas claves se elija como clave **primaria;** a las otras se les llama

entonces claves **alternas.** En el ejemplo ELEMENTO, podríamos elegir {SÍMBOLO} como la clave primaria; {NOMBRE} y {NUMATOMICO} serían entonces las claves alternas. Y en el caso en el que sólo existe una clave candidata, el modelo relacional ha requerido (de nuevo históricamente) que esa clave candidata sea designada como la clave primaria de la varrel base en cuestión. Por lo tanto, toda varrel base siempre ha tenido una clave primaria.

Ahora bien, elegir una clave candidata como primaria podría ser una buena idea en muchos casos (en aquellos casos en los que hay una opción); quizás incluso en la mayoría de los casos, aunque no se puede justificar de manera inequívoca que en *todos* los casos. En la referencia [8.13] se dan los argumentos detallados que apoyan esta postura; aquí, sólo señalamos uno de ellos, el cuál es que la elección de la clave primaria es esencialmente arbitraria (para citar a Codd [8.8], "la base normal [para hacer la elección] es la simplicidad, pero este aspecto está fuera del alcance del modelo relacional"). En nuestros ejemplos, en algunos casos definiremos una clave primaria y en otros no (por supuesto, siempre definiremos por lo menos una clave *candidata).*

Claves externas

En general, una *clave externa* es un conjunto de atributos de una varrel *R2* cuyos valores tienen que coincidir con los valores de cierta clave candidata para cierta varrel *Rl.* Por ejemplo, considere el conjunto de atributos {V#} de la varrel VP (por supuesto, un conjunto que sólo contiene un atributo). Es necesario dejar en claro que debemos permitir que un valor dado de {V#} aparezca en la varrel VP sólo cuando el mismo valor también aparece como un valor de la clave candidata única {V#} de la varrel V (no podemos tener un envío de un proveedor inexistente). En forma similar, debemos permitir que un valor dado del conjunto de atributos {P#} aparezca en la varrel VP sólo cuando el mismo valor aparece también como un valor de la clave candidata única {P#} de la varrel P (tampoco podemos tener un envío de una parte inexistente). Estos ejemplos sirven para fundamentar la siguiente definición:

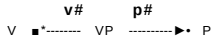
- Sea *R2* una varrel. Entonces, una **clave externa** en *R2* es un conjunto de atributos de *R2* (digamos *FK)* tal que:
  a. Exista una varrel *Rl (Rl* y *R2* no son necesariamente distintas) con una clave candidata *CK,y*
  b. En todo momento, cada valor de *FK* en el valor actual de *R2* sea idéntico al valor de *CK* en alguna tupia del valor actual de *Rl.*

  Puntos a destacar:

1. La definición requiere que cada valor de una clave externa dada aparezca como un valor de la clave candidata correspondiente (la que por lo regular es específicamente, aunque no in variablemente, una clave primaria). Sin embargo, observe que lo opuesto *no* es obligatorio; es decir, la clave candidata correspondiente a una determinada clave externa puede contener un valor que no aparece actualmente como un valor de esa clave externa. Por ejemplo, en el caso de proveedores y partes (la figura 3.8 muestra valores de ejemplo), el número de proveedor V5 aparece en la varrel V, pero no en la varrel VP, ya que el proveedor V5 no suministra actualmente parte alguna,
2. Una clave externa es **simple** o **compuesta** dependiendo del hecho de que la clave candidata con la que coincide sea simple o compuesta.
3. Cada atributo de una clave externa dada debe tener el mismo nombre y tipo que los com ponentes correspondientes de la clave candidata con la que coincide.

4. *Terminología:* Una clave externa representa una **referencia** a la tupia que contiene el valor de clave candidata coincidente (la **tupia referida).** Por lo tanto, al problema de asegurar que la base de datos no incluya ningún valor inválido de clave externa, se le conoce como el problema de **integridad referencial.** A la restricción de que los valores de una clave exter na dada deban coincidir con los valores de la clave candidata correspondiente, se le conoce como **restricción referencia!.** Nos referimos a la varrel que contiene a la clave externa como la varrel **referente** y a la varrel que contiene a la clave candidata correspondiente como la varrel **referida.**
5. *Diagramas referenciales.* Considere una vez más a los proveedores y partes. Podemos repre sentar las restricciones referenciales que existen en esa base de datos por medio del siguiente **diagrama referencial:**

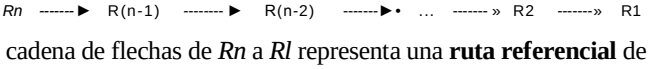
Cada flecha indica que hay una clave externa en la varrel de la que surge la flecha, la cual hace referencia a cierta clave candidata en la varrel a la que apunta la flecha. *Nota:* Por razones de claridad, en ciertas ocasiones es una buena idea etiquetar cada flecha del diagrama referencial con los nombres de atributos que constituyen la clave externa relevante.* Por ejemplo:



Sin embargo, en este libro sólo usaremos dichas etiquetas cuando su omisión pueda conducir a confusión o ambigüedad.

6. Por supuesto, una varrel puede ser referida y referente, como es el caso de *R2* que está a continuación:

Es conveniente introducir el término *ruta referencial*. Sean las relaciones *Rn, R(n-l), ..., R2, Rl* tales que exista una restricción referencial de *Rn* a *R(n-l),* una restricción referencial de *R(n-l)* a *R(n-2), ...,* y una restricción referencial de *R2* a *Rl:*



Entonces, la cadena de flechas de *Rn* a *Rl* representa una **ruta referencial** de *Rn* a *Rl.*

7. Observe que las varrels *Rl* y *R2* en la definición de clave externa anterior *no son necesa riamente distintas.* Es decir, una varrel podría incluir una clave externa cuyos valores nece sitan coincidir con los valores de alguna clave candidata en esa misma varrel. A manera de ejemplo, considere la siguiente definición de varrel (explicaremos la sintaxis en un mo mento, aunque en todo caso debe ser bastante clara por sí misma):

```
VAR EMP BASE RELATION
          { EMP# EMP#, ..., GTE_EMP# EMP#, ... } PRIMARY KEY {
    EMP# } FOREIGN KEY { RENAME GTE_EMP# AS EMP# } REFERENCES EMP
    ;
```

*En forma alternativa (y quizás preferible) podríamos *nombrar* las claves externas y luego usar esos nombres para etiquetar las flechas.

Aquí, el atributo GTE_EMP# representa el número de empleado correspondiente al gerente del empleado identificado por EMP#; por ejemplo, la tupia del empleado E4 podría incluir un valor E3 para GTE__EMP#, el cual representa una referencia a la tupia del empleado E3 en EMP. (Observe en este ejemplo la necesidad de renombrar un atributo de la clave externa a fin de apegarse a los requerimientos del párrafo 3 anterior). En ocasiones se dice que dicha varrel es **autorreferente.** *Ejercicio:* Invente algunos datos para este ejemplo.

Las varrels autorreferentes como EMP, en realidad representan un caso especial de una situación más general; para ser más específicos, pueden existir *ciclos referenciales.* Las varrels *Rn, R(n-l), R(n-2), ..., R2, Rl* forman un **ciclo referencial** cuando*Rn* incluye una clave externa que hace referencia a *R(n-l); R(n-l)* incluye una clave externa que hace referencia a *R(n-2)* y así sucesivamente. Por último, *Rl* incluye una clave externa que hace referencia de nuevo a *Rn.* De manera más breve, un ciclo referencial existe cuando hay una ruta referencial que va de cierta varrel *Rn* hacia ella misma:

R(n-2) **R2** **R1**

9. Algunas veces, se dice que las correspondencias entre claves externas y candidatas son el "pegamento" que mantiene unida a la base de datos. Otra forma de decir lo mismo es que dichas correspondencias representan ciertos *vínculos* entre las tupias. Sin embargo, observe cuidadosamente que no todos esos vínculos están representados por claves de esta forma. Por ejemplo, existe un vínculo ("coubicación") entre proveedores y partes que está representado por los atributos CIUDAD de las relaciones V y P (un proveedor determinado y una parte determinada están coubicados cuando se encuentran en la misma ciudad). No obstante, este vínculo no está representado por claves.

10. De manera histórica, el concepto de clave externa ha sido definido sólo para las varrels base; un hecho que por sí mismo genera algunas preguntas (vea la explicación de *El principio de intercambiabílidad* en el capítulo 9, sección 9.2). Aquí no imponemos dicha restricción; sin embargo, por razones de simplicidad, sí limitamos nuestras explicaciones exclusivamente a las varrels base (donde represente alguna diferencia).

11. El modelo relacional requería originalmente que las claves externas hicieran referencia de manera muy específica a las claves *primarias,* no sólo a las claves candidatas (por ejemplo, vea una vez más la referencia [8.8]). En general, nosotros rechazamos esa limitación por ser innecesaria e indeseable, aunque en la práctica podría constituir a menudo una buena disciplina [8.13]. Por lo regular, en nuestros ejemplos seguiremos esta disciplina.

12. Junto con el concepto de clave externa, el modelo relacional incluye la siguiente regla (la regla de *integridad referencial):*

    ■ **Integridad referencial:** La base de datos no debe contener un valor de clave externa sin correspondencia.*

---

*La integridad referencial puede ser considerada como una *metarrestricción;* esto implica que toda base de datos debe sujetarse a ciertas restricciones de integridad específicas que garanticen que esa base de datos no viole la regla. De paso, observamos que por lo regular se considera que el modelo relacional incluye otra de estas "metarrestricciones": la regla de integridad de la *entidad;* sin embargo, esta última regla tiene que ver con los valores nulos y por lo tanto, posponemos su explicación para el capítulo 18.

Aquí, el término "valor de clave externa sin correspondencia" significa simplemente un valor de clave externa, en alguna varrel referente, para el cual no existe un valor coincidente de la clave candidata relevante en la varrel relevante referida. En otras palabras, la restricción simplemente dice: si *B* hace referencia a *A*, entonces *A* debe existir.

Aquí está la sintaxis para definir una clave externa:

```
FOREIGN KEY { <lista de elementos separados con comas> }
  REFERENCES <nombre de varrel>
```

Esta cláusula aparece dentro de \a definición de una varrel referente. Observe que:

- Cada <*elemento*> puede ser un <*nombre de atributo*> de la varrel referente, o bien una ex presión de la forma

  ```
  RENAME <nombre de atributo> AS <nombre de atributo>
  ```

  (para un ejemplo del caso de RENAME, vea arriba la varrel autorreferente EMP).
- El <*nombre de varrel*> identifica a la varrel referida.

Ya hemos mostrado ejemplos en varios puntos anteriores de este libro (por ejemplo, vea la figura 3.9 del capítulo 3). *Nota:* Por supuesto, como señalamos en la sección 8.5, una definición de clave externa es sólo una forma abreviada de una cierta restricción de base de datos (o de una cierta restricción de varrel, en el caso de una varrel autorreferente), *a menos que* se amplíe la definición de clave externa para que incluya ciertas "acciones referenciales", en cuyo caso se convierte en algo más que sólo una restricción de integridad *por sí misma.* Vea las dos subsecciones que siguen.

Acciones referenciales

Considere la siguiente instrucción:

```
DELETE V WHERE V# = V# ( 'V1' ) ;
```

Suponga que este DELETE hace exactamente lo que dice (es decir, elimina ni más ni menos de los proveedores la tupia del proveedor VI). Suponga también que (a) la base de datos sí incluye algunos envíos del proveedor VI, como en la figura 3.8, y (b) la aplicación no procede a eliminar dichos envíos. Entonces, cuando el sistema verifique la restricción referencial de envíos a proveedores, encontrará una violación y ocurrirá un error.

Nota: Ya que aquí la restricción referencial es una restricción de *base de datos,* la verificación se hará al momento del COMMIT; por lo menos conceptualmente (el sistema podría verificar la restricción tan pronto como se ejecute el DELETE, pero una violación en ese momento no es necesariamente un error, sólo significa que el sistema tendrá que verificar de nuevo al momento del COMMIT).

Sin embargo, existe un enfoque alterno (que podría ser mejor en algunos casos) en el que el sistema realiza una *acción de compensación* correspondiente que garantice que el resultado general satisfaga la restricción. En el ejemplo, la acción de compensación obvia sería que el sistema eliminara "en forma automática" los envíos del proveedor VI. Podemos lograr este efecto si ampliamos la definición de la clave externa como sigue:

```
VAR VP BASE RELATION { ... } ...
    FOREIGN KEY { V# } REFERENCES V
                      ON DELETE CASCADE ;
```

La especificación ON DELETE CASCADE define una *regla de DELETE* para esta clave extema en particular, mientras que la especificación CASCADE es la *acción referencial* de esa regla de DELETE. El significado de estas especificaciones es que una operación DELETE sobre la varrel de proveedores debe actuar "en cascada" para eliminar también las tupias correspondientes en la varrel de envíos.

Otra acción referencial común es RESTRICT (el cual no tiene nada que ver con el operador *restringir* del álgebra relacional). En el caso que nos ocupa, RESTRICT significaría que las operaciones DELETE estarían "restringidas" al caso en donde no existieran envíos que coincidieran (de lo contrario, serían rechazados). Omitir una acción referencial para una clave externa en particular equivale a especificar la "acción" NO ACTION, que significa lo que dice: el DELETE se realiza exactamente como se solicitó, ni más ni menos. (Por supuesto, si en el caso que nos ocupa se especifica NO ACTION y se elimina un proveedor que tiene envíos correspondientes, obtendremos en consecuencia una violación de la integridad referencial.) Los puntos que destacan son:

1. DELETE no es la única operación para la cual las acciones referenciales tienen sentido. Por ejemplo, ¿qué sucedería si intentamos actualizar el número de un proveedor para el que exis te por lo menos un envío que coincide? Está claro que necesitamos una regla para *UPDATE* así como una regla para DELETE. En general, existen las mismas posibilidades para UP DATE que para DELETE:
   - CASCADE. El UPDATE se realiza en cascada para actualizar también la clave externa en aquellos envíos que coincidan;
   - RESTRICT. El UPDATE se restringe al caso en el que no hay envíos que coincidan (en caso contrario, se rechaza);
   - NO ACTION. El UPDATE se realiza exactamente como fue solicitado.
2. Desde luego, CASCADE, RESTRICT y NO ACTION no son las únicas acciones referen ciales posibles; son apenas algunas de las que se requieren comúnmente en la práctica. Sin embargo, en principio podría haber un número arbitrario de respuestas posibles a (por ejem plo) un intento de eliminar un proveedor en particular. Por ejemplo:
   - La información podría ser escrita en alguna base de datos de información histórica;
   - Los envíos del proveedor en cuestión podrían ser transferidos a algún otro proveedor; y así sucesivamente. Nunca será factible proporcionar una sintaxis declarativa para todas las respuestas concebibles. Por lo tanto, en general debe ser posible especificar una acción referencial de la forma "CALL*proc(...)*", en donde*proc* es un procedimiento definido por el usuario.

   Nota: Debe considerar la ejecución de ese procedimiento como parte de la ejecución de la transacción que provocó la verificación de integridad. Además, la verificación de integridad debe volverse a realizar después de ejecutar el procedimiento (obviamente, el procedimiento no debe dejar la base de datos en un estado que viole la restricción).
3. Sean *R2y Rl*, respectivamente, una varrel referente y la varrel referida correspondiente:

Dejemos que la regla de DELETE aplicable especifique CASCADE. Entonces, un DELETE en una tupia dada de *R1* implicará (en general) un DELETE en ciertas tupias de la varrel *R2*. Suponga ahora que a su vez, la varrel *R2* es referida por alguna otra varrel *R3:*

Entonces, el efecto del DELETE implícito sobre las tupias de *R2* se define tal como si se hubiese hecho un intento por eliminar directamente esas tupias; es decir, depende de la regla DELETE especificada para la restricción referencial de *R3* a *R2*. Si ese DELETE implícito falla (debido a la regla de DELETE de *R3* a *R2*, o por cualquier otra razón), entonces la operación completa fracasará y la base de datos permanecerá sin cambio. Y así sucesivamente de manera recursiva, hasta cualquier número de niveles.

Observaciones similares se aplican para la regla de CASCADE UPDATE (haciendo los cambios necesarios) si la clave externa en la varrel *R2* tiene algún atributo en común con la clave candidata de esta varrel, la cual es referida desde la clave externa en *R3*.

4. De lo anterior se desprende que desde un punto de vista lógico, las actualizaciones de bases de datos son siempre atómicas (todo o nada), aunque bajo la superficie involucren diversas actualizaciones sobre distintas varrels (debido, por ejemplo, a una acción referencial CASCADE).

Procedimientos activados

Como quizás se haya dado cuenta (y de hecho, como debió sugerir la observación de la subsección anterior con respecto a los procedimientos definidos por el usuario), todo el concepto de las acciones referenciales nos lleva más allá de las restricciones de integridad como tales y nos introduce en el campo de los *procedimientos activados.* Un **procedimiento activado** (conocido generalmente en la literatura sólo como *activador)* es un procedimiento que es invocado "automáticamente" al ocurrir algún evento o *condición de activación.* Esta condición es típicamente la ejecución de alguna operación de actualización de la base de datos, pero podría ser, por ejemplo, la ocurrencia de una excepción especificada (en particular, la violación de una restricción de integridad especificada) o el transcurso de un intervalo de tiempo especificado. Las acciones referenciales CASCADE ofrecen un ejemplo sencillo de un procedimiento activado (especificado declarativamente, ¡téngalo presente!).

En general, los procedimientos activados se aplican a una variedad mucho más amplia de problemas que la simple cuestión de integridad que es el tema del presente capítulo (en la referencia [8.1] puede encontrar una buena lista de esas aplicaciones). Sin embargo, representan un tema extenso por derecho propio; un tema que está fuera del alcance de este capítulo (para una mejor explicación, vea la referencia [8.22]). Aquí solamente quisiéramos decir que aunque los procedimientos activados son ciertamente útiles para muchos fines, por lo regular *no* son un buen enfoque para el problema específico de la integridad de bases de datos, por razones obvias. (De ser posible, siempre son preferibles los enfoques declarativos.) *Nota:* Estas observaciones no pretenden sugerir que las acciones referenciales sean una mala idea. Aunque es cierto que las acciones referenciales sí ocasionan que ciertos procedimientos sean invocados, al menos son especificadas (como ya lo señalamos) de manera declarativa.

8.9 PROPIEDADES DE SQL

El esquema de clasificación de las restricciones de integridad de SQL es muy diferente al que describimos en las secciones 8.1 a 8.5. Antes que nada, clasifica las restricciones en tres grandes categorías, como sigue:

- Restricciones de dominio
- Restricciones de tablas base
- Restricciones generales ("aserciones")

Sin embargo, las "restricciones de dominio" no son las mismas que nuestras restricciones de tipo, las "restricciones de tablas base" no son las mismas que nuestras restricciones de varrels y las "aserciones" no son las mismas que nuestras restricciones de base de datos. De hecho:

- En realidad, SQL no soporta en absoluto a las restricciones de tipo (debido, por supuesto, a que en realidad no soporta en absoluto *tipos* distintos a unos cuantos tipos integrados).
- Las restricciones de "dominio" de SQL son una forma generalizada —indeseablemente— de nuestras restricciones de atributos (recuerde que los dominios al estilo de SQL en reali dad no son dominios en el sentido relacional).
- Las restricciones de tablas base y las aserciones de SQL (que en efecto son intercambiables) equivalen generalmente a nuestras restricciones de varrel y de base de datos, tomadas en conjunto.

Observamos también que SQL no tiene soporte alguno para las restricciones de transición. Tampoco soporta actualmente procedimientos activados, aunque en SQL3 se incluye parte de este soporte (vea el apéndice **B).**

Restricciones de dominio

Una restricción de dominio al estilo de SQL es una que se aplica a toda columna definida en el dominio en cuestión. El siguiente es un ejemplo:

```
CREATE DOMAIN COLOR CHAR(6) DEFAULT '???'
   CONSTRAINT COLORESJ/ALIDOS CHECK (
   VALUE IN
            ( 'Rojo[1]', 'Amarillo', 'Azul',   'Verde', '???' ) )
```

Suponga que la instrucción CREATE TABLE para la tabla base P, luce como la siguiente:

```
CREATE TABLE P ( ... , COLOR COLOR, ... ) ;
```

Entonces, si el usuario inserta una fila de parte y no proporciona un valor de COLOR para esa fila, se colocará en esa posición el valor "???" en forma predeterminada. Como alternativa, si el usuario *sí* proporciona un valor de COLOR pero no es uno del conjunto legal, la operación fallará y el sistema producirá un diagnóstico adecuado que mencione la restricción COLORES_VALIDOS por su nombre.

Ahora bien, en la sección 8.2 vimos que conceptualmente una restricción de dominio es (o más bien debe ser) sólo una enumeración de los valores que conforman ese dominio; y en este

sentido, el ejemplo de COLORES_VALIDOS cumple de hecho con esta definición. Sin embargo, SQL permite generalmente que una restricción de dominio comprenda una expresión lógica de *complejidad arbitraria;* de ahí que, por ejemplo, los valores válidos de cierto dominio *D* pudiesen depender de los valores que aparecen actualmente en cierta tabla *T*. Quizás le convenga meditar sobre algunas de las implicaciones de esta permisividad injustificada.

Restricciones de tablas base

Cualquiera de las siguientes es una restricción de tabla base de SQL:

- una definición de clave candidata,
- una definición de clave externa,
- una definición de "restricción de verificación".

Enseguida abordaremos cada caso en detalle. *Nota:* Cualquiera de estas definiciones puede estar precedida, de manera opcional, por la frase "CONSTRAINT <*nombre de restricción*>", para proporcionar así un nombre para la nueva restricción (lo mismo es cierto para las restricciones de dominio, como vimos anteriormente en el ejemplo de COLORES_VALIDOS). Por brevedad, omitimos esta opción en los ejemplos siguientes:

Claves candidatas: Una definición de clave candidata toma la forma

```
UNIQUE ( <lista de nombres de columna separados con comas> )
```

o la forma

```
PRIMARY KEY ( <lista de nombres de columna separados con comas> )
```

En ambos casos, la <*lista de nombres de columna separados con comas*> no debe estar vacía. Una tabla base dada puede tener como máximo una especificación PRIMARY KEY, pero cualquier número de especificaciones UNIQUE. En el caso de PRIMARY KEY, damos adicionalmente por hecho que cada columna especificada es NOT NULL, aun cuando NOT NULL no sea especificada de manera explícita (vea más adelante la explicación de restricciones de verificación).

Claves externas: Una definición de clave externa toma la forma

```
FOREIGN KEY ( <lista de nombres de columna separados con comas> )
   REFERENCES <nombre de tabla base>
              [ ( <lista de nombres de columna separados con comas> ) ] [
 ON DELETE <acción referencial> ] [ ON UPDATE <acción referencial> ]
```

en donde <*acción referencial*> es NO ACTION (el valor predeterminado) o CASCADE o SET DEFAULT o SET NULL. Dejamos la explicación de SET DEFAULT y SET NULL para el capítulo 18. La segunda <*lista de nombres de columna separados con comas*> es necesaria cuando la clave externa hace referencia a una clave candidata que no sea una clave primaria. *Nota:* La correspondencia de clave externa a clave candidata se hace sobre la base, no de nombres de columnas, sino de la *posición* de las columnas (de izquierda a derecha) dentro de la listas separadas con comas.

Restricciones de verificación: Una "definición de restricción de verificación" toma la forma

```
CHECK ( <expresión condicional> )
```

Se considera que intentar crear una fila *r* dentro de la tabla base *T,* viola una restricción de verificación para *T* cuando la expresión condicional especificada dentro de esa restricción da como resultado *falsa* para *r. Nota:* En SQL, las expresiones condicionales son el equivalente de lo que en otras partes hemos estado llamando expresiones *lógicas.* En el apéndice A, explico estas expresiones en detalle. Observe en particular que (en el contexto que nos ocupa) la expresión condicional puede ser arbitrariamente compleja; explícitamente, *no* está limitada a una condición que se refiera sólo a *T,* sino que en vez de ello puede referirse a cualquier elemento de la base de datos. Una vez más, quizás le convenga meditar sobre algunas de las implicaciones de esta permisividad injustificada.

Aquí tenemos un ejemplo de CREATE TABLE que comprende restricciones de tablas base de las tres clases:

```
CREATE TABLE VP
     ( V# V# NOT NULL, P# P# NOT NULL, CANT CANT NOT NULL,
       PRIMARY KEY ( V#, P# ), FOREIGN KEY ( V# )
       REFERENCES V
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       FOREIGN KEY ( P# ) REFERENCES P
                          ON DELETE CASCADE
                          ON UPDATE CASCADE,
       CHECK ( CANT > 0 AND CANT < 5001 ) ) ;
```

Aquí, estamos dando por hecho (a) que los dominios V#, P# y CANT ya han sido definidos y (b) que V# y P# han sido explícitamente definidos como claves primarias de las tablas V y P, respectivamente. Además, hicimos deliberadamente uso de la forma abreviada mediante la cual una restricción de verificación de la forma

```
CHECK ( <nombre de columna> IS NOT NULL )
```

puede ser reemplazada por una simple especificación NOT NULL en la definición de la columna en cuestión. Por lo tanto, en el ejemplo sustituimos tres restricciones de verificación un tanto enredadas por tres simples NOT NULLs.

Cerramos esta subsección con una observación sobre una ligera rareza, que es como sigue. Se considera que si la tabla base en cuestión está vacía, *siempre* se satisface una restricción de tabla base de SQL (¡aun cuando la restricción es de la forma "esta tabla no debe estar vacía"!).

Aserciones

En el resto de esta sección nos concentraremos en el tercer caso: las restricciones generales o "aserciones". Las restricciones generales se definen mediante **CREATE ASSERTION;** su sintaxis es:

```
CREATE ASSERTION <nombre de restricción>
       CHECK ( <expresión condicional^ ) ;
```

Y aquí tenemos la sintaxis de **DROP ASSERTION:**

```
DROP ASSERTION <nombre de restricción> ;
```

Observe que a diferencia de todas las demás formas del operador DROP de SQL que explicamos en este libro (DROP DOMAIN, DROP TABLE, DROP VIEW), DROP ASSERTION no ofrece una opción RESTRICT en comparación con CASCADE.

Aquí tenemos algunos ejemplos de CREATE ASSERTION:

1. Todo proveedor tiene por lo menos el status cinco:

```
CREATE ASSERTION RI13 CHECK
       ( ( SELECT MIN ( V. STATUS ) FROM V ) > 4 ) ;
```

2. Toda parte tiene un peso positivo:

```
CREATE ASSERTION RI18 CHECK
       ( NOT EXISTS ( SELECT * FROM P
                      WHERE NOT ( P.PESO > 0.0 ) ) ) ;
```

3. Todas las partes rojas deben estar almacenadas en Londres:

```
CREATE ASSERTION RI99 CHECK
       ( NOT EXISTS ( SELECT * FROM P
                      WHERE  P.COLOR = 'Rojo'
                      AND   P.CIUDAD <> 'Londres' ) ) ;
```

4. Ningún envío tiene un peso total (el peso de las partes por la cantidad enviada) mayor a 20,000:

```
CREATE ASSERTION RI46 CHECK
       ( NOT EXISTS ( SELECT * FROM P, VP WHERE
                      P.P# - VP.P# AND  ( P.PESO
                      * VP.CANT ) >
                                          ■0 ) ) i
```

5. Ningún proveedor con un status menor a 20 puede suministrar parte alguna en una cantidad superior a 500:

```
CREATE ASSERTION RI95 CHECK
       ( NOT EXISTS ( SELECT * FROM V, VP WHERE
                      V.STATUS < 20 AND   V.V#
                      = VP.V# AND   VP.CANT >
                      500 ) ) ;
```

Verificación diferida

El esquema de clasificación de restricciones de integridad de SQL también difiere del nuestro con respecto a la cuestión de en qué momento se realiza la verificación. En nuestro esquema, las restricciones de base de datos son verificadas al momento del COMMIT; las demás son verificadas "de inmediato". En contraste, en SQL las restricciones pueden ser definidas como DEFERRABLE o NOT DEFERRABLE (diferibles o no diferibles); si una restricción dada es DEFERRABLE, además puede ser definida como INITIALLY DEFERRED o INITIALLY IMMEDIATE, lo cual define su estado al principio de cada transacción. Las restricciones NOT DEFERRABLE siempre son verificadas de inmediato, pero las restricciones DEFERRABLE se pueden activar y desactivar dinámicamente por medio de la instrucción

```
SET CONSTRAINTS <lista de nombres de restricción separados con comas> <opción> ;
```

donde <*opción*> puede ser IMMEDIATE o bien DEFERRED. Aquí tenemos un ejemplo:

```
SET CONSTRAINTS RI46, RI95 DEFERRED ;
```

Las restricciones DEFERRABLE sólo son verificadas cuando están en el estado IMMEDIATE. Por supuesto, asignar el estado IMMEDIATE a una restricción DEFERRABLE ocasiona que esa restricción sea verificada de inmediato; si la verificación falla, SET IMMEDIATE también falla. El COMMIT fuerza un SET IMMEDIATE para todas las restricciones DEFERRABLE; si falla entonces cualquier restricción de integridad, la transacción es deshecha.

8.10 RESUMEN

En este capítulo explicamos el concepto crucial de **integridad.** El problema de la integridad consiste en asegurar que los datos contenidos en la base de datos sean *exactos* o *correctos* (y por supuesto, estamos interesados en soluciones **declarativas** para ese problema). De hecho, a esta altura se habrá dado cuenta de que en este contexto "integridad" en realidad significa **semántica.** Son las restricciones de integridad (en particular, los predicados de varrel y de base de datos; vea abajo) las que representan el **significado** de los datos. Y es por ello que, como afirmamos en la sección 8.6, la integridad es de *importancia crucial*

Dividimos las restricciones de integridad en cuatro categorías:

- Una restricción de **tipo** especifica los valores válidos para un tipo (o dominio) dado y es ve rificada durante las invocaciones del **selector** correspondiente.
- Una restricción de **atributo** especifica los valores válidos para un determinado atributo y nunca debe ser violada (dando por hecho que las restricciones de tipo están verificadas).
- Una restricción de **varrel** especifica los valores válidos para una varrel dada y es verificada al **actualizar** la varrel en cuestión.
- Una restricción de **base de datos** especifica los valores válidos para una base de datos dada y es verificada **al momento del COMMIT.**

El AND lógico de todas las restricciones de varrel para una varrel determinada es el **predicado de varrel** (interno) para la misma. Este predicado es el significado de la varrel entendido por el sistema y es el **criterio de aceptación de actualizaciones** sobre esa varrel. *La **Regla** de **oro*** establece que *nunca se permite una operación de actualización que deje a cualquier varrel en un estado que viole su propio predicado.* A su vez, la base de datos está sujeta generalmente a un **predicado de base de datos** y tampoco se permite que una transacción deje a la base de datos en un estado que viole ese predicado.

Después bosquejamos brevemente la idea básica de las restricciones de **transición** (otras restricciones son las de **estado).** Luego explicamos los casos especiales de importancia pragmática de las claves **candidata, primaria, alterna y externa.** Las claves candidatas satisfacen las propiedades de **unicidad** e **irreductibilidad** y toda varrel tiene por lo menos una (¡sin excepciones!). La restricción que indica que los valores de una cierta clave externa deben coincidir con los valores de la clave candidata correspondiente, es una **restricción referencial;** analizamos varias implicaciones de la integridad referencial, incluyendo en particular la noción de **acciones referenciales** (en especial **CASCADE).** Esta última explicación condujo a una breve incursión dentro del campo de los **procedimientos disparados.**

Concluimos nuestras explicaciones con una mirada a los aspectos relevantes de **SQL.** SQL soporta restricciones de "dominio", restricciones de tablas base y "aserciones" (restricciones generales), y el soporte de las restricciones de tablas base comprende el soporte de casos especiales para claves.

EJERCICIOS

8.1 Utilice la sintaxis que presentamos en las secciones 8.2 a 8.5, para escribir restricciones de inte gridad para la base de datos de proveedores, partes y proyectos, como sigue:

a. Las únicas ciudades válidas son Londres, París, Roma, Atenas, Oslo, Estocolmo, Madrid y Amsterdam.

b. Los únicos números de proveedor válidos son aquellos que pueden ser representados mediante una cadena de por lo menos dos caracteres, de los cuales el primero es una "V" y el resto denota un entero entre 0 a 9999.

c. Todas las partes rojas deben pesar menos de 50 libras.

d. Dos proyectos no pueden estar ubicados en la misma ciudad.

e. Como máximo, sólo un proveedor puede estar ubicado en Atenas en cualquier momento.

f. Ningún envío puede tener una cantidad que sea mayor que el doble del promedio de dichas can tidades.

g. El proveedor con el status más alto no debe estar ubicado en la misma ciudad que el de menor status.

h. Todo proyecto debe ubicarse en una ciudad en la que exista por lo menos un proveedor.

i. Todo proyecto debe ubicarse en una ciudad en la que exista por lo menos un proveedor de ese proyecto.

j. Debe existir por lo menos una parte roja, k. El status

promedio de los proveedores debe ser mayor a 18. 1. Todo

proveedor en Londres debe suministrar la parte P2. m. Por lo

menos una parte roja debe pesar menos de 50 libras.

n. Los proveedores en Londres deben suministrar mayor variedad de partes que los proveedores en París.

o. Los proveedores en Londres deben suministrar más partes en total que los proveedores en Pan's.

p. Ninguna cantidad de envío puede ser reducida (en una sola actualización) a menos de la mitad de su valor actual.

q. Los proveedores en Atenas pueden moverse sólo a Londres o a París, y los proveedores en Londres sólo pueden moverse a París.

8.2 Para cada una de sus respuestas al ejercicio 8.1, indique si la restricción es de varrel o de base de datos.

8.3 Para cada una de sus respuestas al ejercicio 8.1, indique las operaciones que podrían ocasionar que se violaran las restricciones aplicables.

8.4 Sean CHAR(5) y CHAR(3) cadenas de caracteres de una longitud de cinco y tres caracteres, res pectivamente. ¿Cuántos tipos existen ahí, uno o dos?

8.5 Sean A y *B* dos varrels. Indique las claves candidatas para cada uno de los siguientes casos:

a. *A* WHERE ...

b. *A {...}*

C. *A* TIMES S

d. *A* UNION B

e. /I INTERSECT B

f. *A* MINUS B

g. *A* JOIN B

h. EXTEND *A* ADD exp AS Z

i. SUMMARIZE *A* PER B ADD *exp* AS Z

j. *A* SEMIJOIN B

k. /I SEMIMINUS *B*

En cada caso, suponga que *Ay B* cumplen los requerimientos de la operación en cuestión (por ejemplo, en el caso de UNION, son del mismo tipo).

8.6 Sea *R* una varrel de grado *n.* ¿Cuál es el número máximo de claves candidatas que puede tener *R*?

8.7 Sea *R* una varrel cuyos únicos valores válidos son las relaciones especiales (y muy importantes) de grado 0, DEE y DUM. ¿Qué claves candidatas tiene *R7*

8.8 En el cuerpo del capítulo explicamos las reglas de DELETE y UPDATE de clave externa, pero no mencionamos una "regla de INSERT" de clave externa. ¿Por qué no?

8.9 Utilice los datos de ejemplo de proveedores, partes y proyectos de la figura 4.5 y diga cuál es el efecto de cada una de las siguientes operaciones:

a. UPDATE proyecto Y7, establece Nueva York para CIUDAD;

b. UPDATE parte P5, establece P4 para P#;

c. UPDATE proveedor V5, establece V8 para V#, si la acción referencial aplicable es RESTRICT;

d. DELETE proveedor V3, si la acción referencial aplicable es CASCADE;

e. DELETE parte P2, si la acción referencial aplicable es RESTRICT;

f. DELETE proyecto Y4, si la acción referencial aplicable es CASCADE;

g. UPDATE envío V1-P1-Y1, establece V2 para V#;

h. UPDATE envío V5-P5-Y5, establece Y7 para Y#;

i. UPDATE envío V5-P5-Y5, establece Y8 para Y#;

j. INSERT envío V5-P6-Y7;

k. INSERT envío V4-P7-Y6;

1. INSERT envío Vl-P2->>yy (en donde *yyy* representa un número de proyecto predeterminado).

8.10 Una base de datos de educación contiene información acerca del esquema de capacitación de una compañía. Para cada curso de capacitación, la base de datos contiene los detalles de todos los cursos de prerrequisito y de todas las ofertas de ese curso; para cada oferta, contiene los detalles de todos los maestros y de todos los estudiantes inscritos. La base de datos también contiene información acerca de los empleados. Las varrels relevantes son las siguientes (a grandes rasgos):

```
CURSO        { CURSO#, TITULO }
PRERREQ      { CURSO_SUP#, CURSO_SUB# }
OFERTA       { CURSO*, OF#, FECHAOF, UBICACIÓN }
MAESTRO      { CURSO*, OF#, EMP* }
INSCRIPCIÓN  { CURSO*, OF#, EMP*, CALIF }
EMPLEADO     { EMP#, NOMEMP, PUESTO }
```

El significado de la varrel PRERREQ es que el curso superior (CURSO_SUP#) tiene el curso subordinado (CURSO_SUB#) como un prerrequisito inmediato; las demás deben ser muy claras. Dito un diagrama referencial para esta base de datos. Proponga también la definición correspondiente la base de datos (es decir, escriba un conjunto adecuado de definiciones de tipo y de varrel).

8.11 Las dos siguientes varrels representan una base de datos que contiene información acerca de departamentos y empleados:

```
DEPTO  { DEPT0#,     .,
EMP    { EMP#, ..    GTE_EMP#,
                     DEPTO#, ...
```

Todo departamento tiene un gerente (GTE_EMP#); todo empleado tiene un departamento (DEPT0#) Una vez más, dibuje un diagrama referencial y escriba una definición de base de datos para e base.

8.12 Las dos siguientes varrels representan una base de datos que contiene información acerca de empleados y programadores:

```
EMP    { EMP#,    . ., PUESTO, .. . }
PGMR   { EMP#,    .., LGJE, ... }
```

Cada programador es un empleado, pero no al revés. Una vez más, escriba un diagrama refere escriba una definición adecuada de base de datos.

8.13 Un aspecto que no explicamos en el cuerpo del capítulo es la cuestión de qué debe sucederá usuario intenta eliminar alguna varrel o tipo, así como cierta restricción existente que se refiere a esa varrel o tipo. ¿Qué *debe* suceder en esta situación?

8.14 Proponga soluciones de SQL al ejercicio 8.1.

8.15 Compare y contraste el soporte de la integridad en SQL con el mecanismo de integridad **descrito** en el cuerpo del capítulo.

REFERENCIAS Y BIBLIOGRAFÍA

8.1 Alexander Aiken, Joseph M. Hellerstein y Jennifer Widom: "Static Analysis Techniques for Prt dieting the Behavior of Active Database Rules", *ACM TODS 20,* No. 1 (marzo, 1995).

Este artículo contiene el trabajo de las referencias [8.2] y [8.5] sobre "sistemas de base de dato! expertos" (denominados aquí sistemas de base de datos *activos).* En particular, describe el si¡ tema de reglas del prototipo IBM Starburst (vea las referencias [17.50], [25.14], [25.17] y **[25.21** a 25.22], además de la referencia [8.22]).

8.2 Elena Baralis y Jennifer Widom: "An Algebraic Approach to Rule Analysis in Expert Database Systems", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

De acuerdo con este artículo, un "sistema de base de datos experto" es un sistema de base de datos que soporta "reglas de condición/acción" (en nuestra terminología, la condición es uní *condición de disparo* y la acción es el *procedimiento disparado* correspondiente). Un problema con estos sistemas es que su comportamiento es, inherentemente, difícil de predecir o entender.

Este artículo presenta métodos para determinar (antes de la ejecución) si un conjunto de reglas dado posee las propiedades de *terminación* y *confluencia. Terminación* significa que se garantiza que el proceso de reglas no continuará indefinidamente. *Confluencia* significa que el estado final de la base de datos es independiente del orden en el que se ejecutan las reglas.

8.3 Philip A. Bernstein, Barbara T. Blaustein y Edmund M. Clarke: "Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data", Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada (octubre, 1980).

Presenta un método eficiente para hacer cumplir las restricciones de integridad de una clase especial. Un ejemplo es "todo valor en el conjunto *A* debe ser menor que todo valor en el conjunto *B".* La técnica para hacer cumplir la restricción se basa en la observación de que (por ejemplo) la restricción que acabamos de dar es lógicamente equivalente a la restricción "el valor *máximo* en *A* debe ser menor al valor *mínimo* en *B".* Al reconocer esta clase de restricción, y al mantener en forma automática los valores máximo y mínimo relevantes en las variables ocultas, el sistema puede reducir a *una* el número de comparaciones implicadas en el cumplimiento de la restricción sobre una actualización dada, de una cantidad que está en el orden de cardinalidad de A o de *B* (dependiendo del conjunto al que se aplique la actualización); por supuesto, al costo de tener que mantener las variables ocultas.

8.4 O. Peter Buneman y Erik K. demons: "Efficiently Monitoring Relational Databases", *ACM TODS 4,* No. 3 (septiembre, 1979).

Este artículo se ocupa de la implementación eficiente de los procedimientos disparados (llamados aquí *alertadores);* para el problema particular de decidir cuándo se satisface la condición de disparo, sin evaluar necesariamente la condición. Ofrece un método (un algoritmo de *elusion)* para detectar actualizaciones que posiblemente no pueden satisfacer una determinada condición de disparo; también explica una técnica para reducir la sobrecarga de procesamiento, en el caso en que el algoritmo de elusion fallara, mediante la evaluación de la condición de disparo para un subconjunto pequeño (un *filtro)* del conjunto total de tupias relevantes.

8.5 Stefano Ceri y Jennifer Widom: "Deriving Production Rules for Constraint Maintenance", Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (agosto, 1990).

Describe un lenguaje basado en SQL para definir restricciones y da algoritmos para identificar todas las operaciones que podrían violar una restricción dada. (Un esquema preliminar de dicho algoritmo se dio antes en la referencia [8.11]. La existencia de ese algoritmo significa que no hay necesidad de indicar al DBMS, de manera explícita, cuándo necesita ser verificada una restricción dada y por supuesto, el esquema que describimos en el cuerpo de este capítulo no ofrece ninguna forma de que el usuario pueda hacerlo.) El artículo aborda también las cuestiones de optimization y corrección.

8.6 Stefano Ceri, Piero Fraternali, Stefano Paraboschi y Letizia Tanca: "Automatic Generation of Pro duction Rules for Integrity Maintenance", *ACM TODS 19,* 3 (septiembre, 1994).

Este artículo, basado en el trabajo de la referencia [8.5], presenta la posibilidad de *reparar automáticamente* el daño causado por una violación a una restricción. Las restricciones se compilan dentro de *reglas de producción* con los componentes siguientes:

1. Una lista de operaciones que pueden violar la restricción;

2. Una expresión lógica que dará como resultado *verdadero* si se viola la restricción (en esencia, es sólo la negación de la restricción original);

3. Un procedimiento SQL de reparación.

El artículo incluye también una buena investigación de trabajos relacionados.

8.7 Roberta Cochrane, Hamid Pirahesh y Nelson Mattos: "Integrating Triggers and Declarative Con traints in SQL Database Systems", Proc. 22nd Int. Conf. on Very Large Data Bases, Mumbai (Bombay!. India (septiembre, 1996).

Para citar: "Se debe definir con claridad la semántica de la interacción de los disparadores y restricciones declarativas, para evitar una ejecución inconsistente y proporcionar a los usuarios un modelo amplio para entender dichas interacciones. Este [artículo] define dicho modelo", modelo en cuestión se implemento en DB2 y es "aceptado como el modelo del estándar re tante de SQL (SQL3)" (vea el apéndice B).

8.8 E. F. Codd: "Domains, Keys, and Referential Integrity in Relational Databases", *InfoDB 3*, No. 1 (Primavera de 1988).

Una explicación de los conceptos de dominio, clave primaria y clave externa. El artículo es obviamente confiable, ya que Codd fue el inventor de los tres conceptos; sin embargo, en mi opini deja aún sin resolver o explicar muchos aspectos. Por cierto, ofrece el siguiente argumento a f de la disciplina de seleccionar una clave candidata como primaria: "Omitir el soporte de esta di ciplina es algo así como tratar de usar una computadora sin un esquema de direccionamiento... qu cambia la base cada vez que ocurre una clase de evento en particular (por ejemplo, al encontrar dirección que resulta ser un número primo)". Pero si aceptamos este argumento, ¿por qué no vario hasta su conclusión lógica y usar un esquema de direccionamiento idéntico para to es difícil tener que "direccionar" los proveedores por número de proveedor y las partes pornún» de parte?; por no mencionar los envíos que comprenden "direcciones" que son *compuestas.* (De hecho, hay mucho qué decir de esta idea de un esquema de direccionamiento globalmente uniforme. Vea la explicación de *sustitutos* en la nota a la referencia [13.16] del capítulo 13.)

8.9 C. J. Date: "Referential Integrity", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, Franct (septiembre, 1981). Reeditado en forma revisada en *Relational Database: Selected Writings.* Reading, Mass.: Addison-Wesley (1986).

El artículo que introdujo las acciones referenciales (principalmente CASCADE y RESTRICT que explicamos en la sección 8.8 del presente capítulo. *Nota:* La diferencia principal entre la versión original (VLDB 1981) del artículo y la versión revisada, es que la primera, siguiendoa referencia [13.6], permitía que una clave externa dada hiciera referencia a cualquier cantidad i varrels, mientras que la versión revisada se apartó de esa postura general excesiva por las razón que explicamos en detalle en la referencia [8.10].

8.10 C. J. Date: "Referential Integrity and Foreign Keys" (en dos partes), en *Relational Da Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

La Parte I de este artículo expone la historia del concepto de integridad referencial y ofrece ui conjunto preferido de definiciones básicas (con razonamiento). La Parte II proporciona más a mentos a favor de esas definiciones preferidas y da algunas recomendaciones prácticas esp© ficas; en particular, explica problemas causados por (a) un traslape de claves externas, (b) valores parcialmente nulos de claves externas compuestas, y (c) rutas referenciales *semejantes* (es dea rutas referenciales distintas que tienen el mismo punto de partida y el mismo punto final). JVoi Algunas de las posturas de este artículo están ligeramente influenciadas (aunque no demasiad! por los argumentos de la referencia [8.13].

8.11 C. J. Date: "A Contribution to the Study of Database Integrity", en *Relational Database Writii 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Para citar del resumen: "Este artículo pretende imponer cierta estructura en el problema [déla integridad] mediante (a) la proposición de un esquema de clasificación para las restricciones de integridad, (b) la utilización de ese esquema para clarificar los principales conceptos subyacentes de la integridad de los datos, (c) el bosquejo de un enfoque con un lenguaje concreto

formular restricciones de integridad, y (d) el señalamiento de algunas áreas específicas para investigación posterior". Partes de este capítulo están basadas en este artículo anterior, pero el propio esquema de clasificación debe ser tomado como superado por la versión revisada que describimos en las secciones 8.2 a 8.5 del presente capítulo.

8.12 C. J. Date: "Integrity", capítulo 11 de C. J. Date y Colin J. White, *A Guide to DB2* (4a. edición) [4.20]. Reading, Mass.: Addison-Wesley (1993).

El producto DB2 de IBM sí ofrece el soporte declarativo de claves primarias y externas (de hecho, fue uno de los primeros en hacerlo, si no es que *el primero).* Sin embargo, como explica esta referencia, el soporte sufre por ciertas restricciones de implementación cuya finalidad general es *garantizar el comportamiento predecible.* Damos aquí un ejemplo sencillo. Suponga que la varrel *R* contiene actualmente sólo dos tupias con los valores 1 y 2, respectivamente, en la clave primaria; considere también la solicitud de actualización "Duplicar todo valor de clave primaria de *R".* El resultado correcto es que ahora las tupias tienen los valores 2 y 4 en la clave primaria. Si el sistema actualiza primero el "2" (reemplazándolo por "4") y después actualiza el "1" (remplazándolo por "2"), la solicitud tendrá éxito. Si, por otra parte, el sistema actualiza —o más bien, intenta actualizar— primero el "1" (reemplazándolo por "2"), caerá en una violación de la unicidad y la solicitud fracasará (la base de datos permanecerá sin cambio). En otras palabras, *el resultado de la consulta es imprevisible.* Con el fin de evitar esa falta de previsibilidad, DB2 simplemente evita situaciones en las que, de lo contrario, ésta podría ocurrir. Sin embargo, algunas de las restricciones resultantes son por desgracia bastante severas [8.17]

Obseve que, como sugiere el ejemplo anterior, DB2 realiza por lo general una "verificación sobre la marcha"; es decir, aplica las verificaciones de integridad a cada tupia individual *al actualizar esa tupia.* Esta verificación sobre la marcha es lógicamente incorrecta (vea la explicación de las operaciones de actualización al final de la sección 5.4 del capítulo 5). Esto se hace por motivos de rendimiento.

8.13 C. J. Date: "The Primacy of Primary Keys: An Investigation", en *Relational Database Writings 1991-1994.* Reading, Mass.: Addison-Wesley (1995).

Presenta argumentos en apoyo a la postura de que a veces no es una buena idea hacer que una clave candidata sea "más igual que otras".

8.14 M. M. Hammer y S. K. Sarin: "Efficient Monitoring of Database Assertions", Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas (mayo/junio, 1978).

Se bosqueja un algoritmo para generar procedimientos de verificación de integridad que sean más eficientes que el método obvio de la "fuerza bruta", que simplemente evalúa las restricciones después de realizar una actualización. Las verificaciones se incorporan al código objeto de la aplicación en tiempo de compilación. En algunos casos, es posible detectar que no se requieren en absoluto verificaciones en tiempo de ejecución. Con frecuencia es posible reducir de manera importante el número de accesos a la base de datos (de formas muy diversas).

8.15 Bruce M. Horowitz: "A Run-Time Execution Model for Referential Integrity Maintenance", Proc. 8th IEEE Int. Conf. on Data Engineering, Phoenix, Ariz, (febrero, 1992).

Es bien sabido que ciertas combinaciones de

1. Estructuras referenciales (es decir, colecciones de varrels que se interrelacionan mediante res tricciones referenciales),
2. Reglas de DELETE y UPDATE de clave externa, y
3. Valores de datos reales en la base de datos,

pueden conducir a ciertas situaciones de conflicto y causar potencialmente un comportamiento impredecible por parte de la implementación (vea por ejemplo la referencia [8.10] para una mayor explicación). Existen tres grandes enfoques para enfrentar este problema:

a. Dejarlo al usuario;

b. Hacer que el sistema detecte y rechace los intentos de definir estructuras que pudieran conduc a conflictos potenciales en tiempo de ejecución; o

c. Hacer que el sistema detecte y rechace conflictos *reales* en tiempo de ejecución.

La opción a. está destinada al fracaso y la opción b. tiende a ser precavida en extremo [8.12. 8.17]. Por lo tanto, Horowitz propone la opción c. El artículo ofrece un conjunto de reglas pan dichas acciones en tiempo de ejecución y demuestra que son correctas. Sin embargo, observe quí no considera la sobrecarga en el rendimiento de dicha verificación en tiempo de ejecución. Horowitz fue un miembro activo del comité que definió el SQL/92, y las partes de integric referencial de ese estándar implican en efecto que hay que manejar las propuestas de este artículo

8.16 Victor M. Markowitz: "Referential Integrity Revisited: An Object-Oriented Perspective". Prix 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (agosto, 1990).

La "perspectiva orientada a objetos" que aparece en el título de este artículo, refleja la postura inicial del autor de que "la integridad referencial subyace en la representación relacional de estructuras orientadas a objetos". Sin embargo, el artículo en realidad no trata sobre la orienta' a objetos. Más bien, presenta un algoritmo que, a partir de un diagrama entidad/vínculo capítulo 13), generará una definición de base de datos relacional, en la que se garantiza que no ocurrirán algunas de las situaciones problemáticas identificadas en la referencia [8.10] (por ejemplo, el traslape de claves).

El artículo explica también tres productos comerciales (DB2, Sybase e Ingres, de alrededor de 1990) desde el punto de vista de la integridad referencial. DB2, el cual proporciona un soporte *declarativo,* se muestra como excesivamente restrictivo; Sybase e Ingres, los cuales proporcioni un soporte *de procedimientos* (mediante "disparadores" y "reglas", respectivamente), se muestran menos restrictivos que DB2 pero resultan enredados y difíciles de utilizar (aunque se dice que el soporte de Ingres es "técnicamente superior" al de Sybase).

8.17 Victor M. Markowitz: "Safe Referential Integrity Structures in Relational Databases", Proc. 17tl Int. Conf. on Very Large Data Bases, Barcelona, Spain (septiembre, 1991).

Propone dos "condiciones de seguridad" formales que garantizan que no ocurran algunas situaciones problemáticas que explicamos en las referencias [8.10] y [8.15] (por ejemplo). El artículo considera también lo que implica satisfacer dichas condiciones en DB2, Sybase e Ingres (de nuevo, alrededor de 1990). Con respecto a DB2, el artículo muestra que algunas de las restricciones de implementación impuestas en interés de la seguridad (vea la referencia [8.12]) son lógicamente innecesarias, mientras que al mismo tiempo otras son inadecuadas (es decir, DB2 todavía permite ciertas situaciones inseguras). Con respecto a Sybase e Ingres, afirma que el soporte de procedimientos que se encuentra en esos productos no ofrece la detección de especificaciones referenciales inseguras ¡o incluso incorrectas!

8.18 Ronald G. Ross: *The Business Rule Book: Classifying, Defining, and Modeling Rules* (Version 3.0). Boston, Mass.: Database Research Group (1994).

Vea la nota a la referencia [8.19].

8.19 Ronald G. Ross: *Business Rule Concepts.* Houston, Tx.: Business Rule Solutions Inc. (1998).

Durante los últimos años, se ha generado en el mundo comercial un buen número de reglas del **negocio;** algunas figuras de la industria han comenzado a sugerir que se tome a éstas como base para diseñar y generar bases de datos y aplicaciones (es decir, en vez de las técnicas más establecidas como el modelado "entidad/vínculo", "el modelado de objetos", "el modelado semántico" y otros). Y nosotros estamos de acuerdo porque las "reglas del negocio" en esencia no son más que una forma más sencilla (es decir, menos académica y menos formal) de hablar sobre los predicados,

proposiciones y todos los demás aspectos de integridad que explicamos en el presente capítulo. Ross es uno de los principales defensores del enfoque de reglas del negocio y recomiendo su libro para todo practicante serio. La referencia [8.18] es amplia, la referencia [8.19] es un tutorial breve.

8.20 M. R. Stonebraker y E. Wong: "Access Control in a Relational Data Base Management System by Query Modification", Proa ACM National Conf. (1974).

El prototipo University Ingres [7.11] fue pionero de un enfoque interesante para las restricciones de integridad (y también para las restricciones de seguridad; vea el capítulo 16), el cual está basado en la *modificación de solicitudes.* Las restricciones de integridad fueron definidas por medio de la instrucción DEFINE INTEGRITY, cuya sintaxis es:

```
DEFINE INTEGRITY ON <nombre de varrel> IS <expresión lógica>
```

Por ejemplo:

```
DEFINE  INTEGRITY ON V  IS V.STATUS >  0
```

Suponga que un usuario *U* intenta la siguiente operación REPLACE de QUEL:

```
REPLACE V  ( STATUS  = V.STATUS  -  10  )
WHERE     V.CIUDAD =  "Londres"
```

Luego Ingres modifica automáticamente el REPLACE como:

```
REPLACE V ( STATUS ■ V.STATUS - 10 )
WHERE  V.CIUDAD = "Londres" AND    (
V.STATUS - 10 ) > 0
```

Y por supuesto, no es posible que esta operación modificada viole la restricción de integridad. Una desventaja de este enfoque es que no todas las restricciones pueden hacerse cumplir de esta manera sencilla; de hecho, QUEL soportaba sólo restricciones en las que la expresión lógica era una condición de restricción simple. Sin embargo, ese soporte limitado representó más de lo que había en la mayoría de los sistemas en ese momento.

8.21 A. Walker y S. C. Salveter: "Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates", State University of New York, Stony Brook, N.Y.: Technical Report 81/026 (junio, 1981).

Describe una técnica para modificar cualquier "plantilla de transacción" (es decir, código fuente de transacción) en una plantilla *segura* correspondiente; segura, en el sentido de que ninguna instancia de transacción que se apegue a esa plantilla modificada puede violar ninguna restricción de integridad declarada. El método funciona agregando consultas y comprobaciones a la plantilla original para asegurar que *antes* de realizar cualquier actualización, no se esté violando ninguna restricción. Si falla cualquiera de estas comprobaciones en tiempo de ejecución, la transacción será rechazada y se generará un mensaje de error.

8.22 Jennifer Widom y Stefano Ceri (eds.): *Active Database Systems: Triggers and Rules for Advanced Database Processing.* San Francisco, Calif.: Morgan Kaufmann (1996).

Un útil compendio de artículos de investigación y tutoriales sobre "sistemas de base de datos activos" (es decir, sistemas de base de datos que realizan automáticamente acciones específicas en respuesta a eventos específicos; en otras palabras, sistemas de base de datos con procedimientos disparados). Incluye las descripciones de varios sistemas prototipo, en particular el Starburst de IBM Research (vea las referencias [17.50], [25.14], [25.17] y [25.21 a 25.22]) y el Postgres de la Universidad de

RESPUESTAS A EJERCICIOS SELECCIONADOS

8.1

```
a. TYPE CIUDAD POSSREP (CHAR )
   CONSTRAINT THE_CIUDAD (  CIUDAD ) =  'Londres'
           OR THE_CIUDAD (  CIUDAD ) =  'París'
           OR THE_CIUDAO (  CIUDAD ) "  'Roma'
           OR THE_CIUDAD (  CIUDAD ) =  'Atenas'
           OR THE_CIUDAD (  CIUDAD ) •  'Oslo'
           OR THE_CIUDAD (  CIUDAD ) ■  'EstOCOlmo'
           OR THE_CIUDAD (  CIUDAD ) =  'Madrid'
           OR THECIUDAD (  CIUDAD ) =  'Amsterdam' ;
```

Una forma abreviada obvia sena:

```
a.  TYPE CIUDAD POSSREP ( CHAR )
    CONSTRAINT THE_CIUDAD ( CIUDAD ) IN { 'Londres',  'París'
                                          'Roma'  ,  'Atenas'    ,
                                          'Oslo'  ,  'Estocolmo' ,
                                          'Madrid' ,  'Amsterdam' } ;

b.  TYPE V# POSSREP ( CHAR ) CONSTRAINT
        SUBSTR ( THE_V# | VI ), 1, t ) • '¥' AND CAST_AS_INTEGER
        ( SUBSTR ( THE_V# ( V# ), 2 ) ) > 0 AND CAST_AS_INTEGER (
        SUBSTR ( THE_V# ( V# ) , 2 ) ) < 9999 ;
```

Damos aquí por hecho que tanto el operador de subcadena SUBSTR como el operador de conversión explícita CAST_AS_INTEGER están disponibles.

```
C. CONSTRAINT C IS_EMPTY ( P WHERE PESO > PESO ( 50.0 ) ) ;

d.  CONSTRAINT D COUNT ( Y ) = COUNT ( Y { CIUDAD } ) ;

e.  CONSTRAINT E COUNT ( V WHERE CIUDAD = 'Atenas' ) < 1 ;

f.  CONSTRAINT F
       IS_EMPTY ( ( EXTEND VPY ADD 2 * AVG ( VPY, CANT )
                                AS X ) WHERE CANT > X ) ;

    CONSTRAINT G
       IS_EMPTY ( ( V WHERE STATUS = MIN   V { STATUS } ) ) JOIN
                  ( V WHERE STATUS - MAX   V { STATUS } ) ) ) ;
```

En realidad, los términos "proveedor con el más alto status" y "proveedor con el más bajo satus" no están bien definidos, ya que los valores de status no son únicos. Interpretamos el» querimiento como que si *Vx* y *Vy* son *cualquier* proveedor con el "status más alto" y "el sta más bajo", respectivamente, entonces *Vx* y *Vy* no deben estar coubicados. Observe que la restric ción será necesariamente violada si el status "más alto" y el "más bajo" son iguales (en partí lar, se violará si sólo existe un proveedor).

```
h. CONSTRAINT H IS_EMPTY ( Y { CIUDAD } MINUS V { CIUDAD } ) ;

i.  CONSTRAINT I IS_EMPTY ( Y WHERE NOT ( TUPLE { CIUDAD CIUDAD } IN
                          ( Y { Y# } JOIN VPY JOIN V ) { CIUDAD } ) ) ;

j.  CONSTRAINT J NOT ( IS_EMPTY
                     ( WHERE COLOR ■ COLOR ( 'Rojo' ) ) ) ;
```

Esta restricción será violada si no existe parte alguna. Una mejor formulación sería:

```
CONSTRAINT J IS_EMPTY ( P ) OR NOT ( IS_EMPTY
               ( P WHERE COLOR = COLOR ( 'Rojo' ) ) ) ;

k. CONSTRAINT K IF NOT ( IS_EMPTY ( V ) )
                  THEN AVG ( V, STATUS ) > 18
               END IF ;
```

Aquí, la comprobación IF es para evitar el error que de otra forma ocurriría si el sistema tratara de verificar la restricción cuando no hubiese proveedor alguno.

```
 1. CONSTRAINT L IS_EMPTY
                ( ( V WHERE CIUDAD = 'Londres' ) { V# } MINUS (
                  VPY WHERE P# = P# ( 'P2' ) ) { V# } ) ;

m. CONSTRAINT M IS_EMPTY ( P ) OR NOT
               ( IS_EMPTY ( P WHERE PESO < PESO ( 50.0 ) ) ) ;

 n. CONSTRAINT N
    COUNT ( ( ( V WHERE CIUDAD = 'Londres' ) JOIN VPY ) { P# } ) >
    COUNT ( ( ( V WHERE CIUDAD = 'París'   ) JOIN VPY ) { P# } ) ;

 O. CONSTRAINT 0
        SUM ( ( ( V WHERE CIUDAD = 'Londres'  ) JOIN VPY ), CANT )
        SUM ( ( ( V WHERE CIUDAD = 'París'    ) JOIN VPY ), CANT )

 p. CONSTRAINT P IS_EMPTY
               ( ( VPY JOIN ( VPY' RENAME CANT AS CANT' ) )
                 WHERE CANT > 0.5 * CANT' ) ;

 q. CONSTRAINT 0 IS_EMPTY (
      ( V JOIN ( V WHERE CIUDAD • 'Atenas' ) ) WHERE CIUDAD * 'Atenas'
        AND CIUDAD * 'Londres' AND CIUDAD ■ 'París' )
               AND IS_EMPTY (
      ( V JOIN ( V WHERE CIUDAD = 'Londres' ) ) WHERE
        CIUDAD * 'Londres' AND CIUDAD ■ 'París' ) ;
```

Como ejercicio adicional, podría intentar formular las restricciones anteriores en el estilo del cálculo en lugar del estilo del álgebra.

8.2 Por supuesto, las dos primeras son restricciones de tipo. Con respecto a las demás, las restricciones C, D, E, F, G, J, K, M, P y Q son restricciones de varrel y el resto son restricciones de base de datos.

8.3

a. Invocación al selector CIUDAD.
b. Invocación al selector V#.
c. INSERT sobre P, UPDATE sobre PESO de partes.
d. INSERT sobre Y, UPDATE sobre CIUDAD de proyectos.
e. INSERT sobre V, UPDATE sobre CIUDAD de proveedores.
f. INSERT o DELETE sobre VPY, UPDATE sobre CANT de envíos.
g. INSERT o DELETE sobre V, UPDATE sobre STATUS de proveedores.
h. INSERT sobre Y, DELETE sobre V, UPDATE sobre CIUDAD de proveedores o proyectos, i. INSERT sobre Y, DELETE sobre VPY, UPDATE sobre CIUDAD de proveedores o proyectos.

j. INSERT o DELETE sobre P, UPDATE sobre CIUDAD de partes.

k. INSERT o DELETE sobre V, UPDATE sobre STATUS de proveedores.

1. INSERT sobre V, DELETE sobre VPY, UPDATE sobre CIUDAD de proveedores o sobre Y« o P# de envíos.

m. INSERT o DELETE sobre P, UPDATE sobre PESO de partes, n. INSERT o DELETE sobre V o VPY, UPDATE sobre CIUDAD de proveedores o sobre VII P# de envíos.

0. INSERT o DELETE sobre V o VPY, UPDATE sobre CIUDAD de proveedores o sobre V# o P# o CANT de envíos.

p. UPDATE sobre CANT de envíos.

q. UPDATE sobre CIUDAD de proveedores.

8.4 Uno. Las especificaciones "(5)" y "(3)" se ven mejor como *restricciones de integridad.* Como lamos en la referencia [3.3], una consecuencia deseable de este enfoque es que las variables X y Ys definidas, digamos, como CHAR(5) y CHAR(3), respectivamente. Entonces las comparaciones entre y Y son válidas; es decir, no violan el requerimiento de que los operandos deben ser del mismo tip

8.5 Ofrecemos las siguientes como un "primer corte" del conjunto de respuestas (pero vea la nota a final).

a. Toda restricción de *A* hereda todas las claves candidatas de *A.*

b. Si la proyección incluye cualquier clave candidata *K* de *A,* entonces *K* es una clave candidata la proyección. En caso contrario, la única clave candidata es la combinación de todos los butos de la proyección (en general).

c. Toda combinación *K* de una clave candidata *KA* de A y una clave candidata *KB* de *B,* es una < candidata del producto *A* TIMES *B.*

d. La única clave candidata para la unión *A* UNION *B* es la combinación de todos los atributos! general).

e. La dejamos como un ejercicio (la intersección no es una primitiva).

f. Toda clave candidata de *A* es una clave candidata de la diferencia *A* MINUS *B.*

g. Dejamos el caso general como un ejercicio (la junta natural no es una primitiva). Sin emb subrayamos que en el caso especial donde el atributo de junta en *A* es una clave candidata i toda clave candidata de *B* es una clave candidata de la junta.

h. Las claves candidatas de una extensión cualquiera de *A* son las mismas que las claves candidatas de A

1. Las claves candidatas de un resumen cualquiera de *A* "por *B"* son las claves candidatas d

j. Toda clave candidata de *A* es una clave candidata de la semijuntaA SEMIJOIN *B.*

k. Toda clave candidata de *A* es una clave candidata de la semidiferencia *A* SEMIMINUS

Sin embargo, en ciertas situaciones muchas de las instrucciones anteriores pueden refinai alguna forma. Por ejemplo:

- La combinación {V#,P#,Y#j no es una clave candidata para la restricción VPY WHERE V# = V#('VI'); más bien, lo es la combinación {P#,Y#};
- Si *A* tiene un encabezado *[X, Y,Z)* y sólo la clave candidata *X* satisface la dependencia funcii *Y* —> Z (vea el capítulo 10), entonces *Y* es una clave candidata de la proyección de *A* sobre *Y)*
- Si *A* y *B* son ambas restricciones de *C,* entonces toda clave candidata de C es una clave caí data de A UNIONS;

y así sucesivamente. Toda la cuestión de la *inferencia de claves candidatas* se explica con cierto talle en la referencia [10.6].

8.6 Sea *m* el entero mayor o igual que *n/2*. *R* tendrá el máximo número posible de claves candidatas si (a) todo conjunto distinto de atributos de *m* es una clave candidata, o bien (b) *n* es non y cada con junto distinto de *m*-\ atributos es una clave candidata. De cualquier forma, se desprende que el número máximo de claves candidatas en *R* es *n! / (m! * (n-m)!*). *Nota:* Las varrels ELEMENTO y MATRI MONIO de la sección 8.8 son ejemplos de varrels con el máximo número posible de claves candidatas.

8.7 *R* tiene exactamente una clave candidata; es decir, el conjunto vacío de atributos {} (en ocasiones escrito como (o)). *Nota:* El concepto de una clave candidata vacía (o *nula)* bien vale para abundar un poco. Una varrel como *R* cuyos únicos valores válidos son DEE y DUM deben necesariamente tener ningún atributo y por lo tanto es "obvio" que su única clave candidata tampoco tiene atributos. Pero no sólo las varrels sin atributos pueden tener una clave candidata así. Sin embargo, si $ es una clave candidata de cierta varrel *R,* entonces:

- Ésta debe ser la *única* clave candidata de *R,* ya que cualquier otro conjunto de atributos de *R* sería un superconjunto propio de <|), y por lo tanto violaría el requerimiento de irreductibilidad de las claves candidatas. (Por lo tanto se trata de hecho de la *clave primaria,* si debe elegirse una.)
- *R* está restringida a contener como máximo una tupia, puesto que cada tupia tiene el mismo valor (es decir, la 0-tupla) para el conjunto vacío de atributos.

Observe que nuestra sintaxis sí permite la declaración de una varrel así; por ejemplo:

```
VAR R BASE RELATION { ... }
    PRIMARY KEY { } ;
```

También permite declarar una varrel sin atributo alguno, es decir, una varrel cuyos únicos valores posibles son DEE y DUM:

```
VAR R BASE RELATION { }
    PRIMARY KEY { } ;
```

Para retomar la cuestión de claves candidatas vacías, debemos decir que (desde luego) si una clave candidata puede estar vacía, también puede estarlo una clave externa correspondiente. La referencia [5.5] explica esta posibilidad con cierto detalle.

8.8 No hay una "regla INSERT" explícita de clave externa debido a que los INSERT en la varrel referente (así como los UPDATE sobre la clave externa en la varrel referente) están gobernados por la propia regla de integridad referencial básica; es decir, por el requerimiento de que no haya valores de claves externas sin correspondencia. En otras palabras, tomando como ejemplo concreto a provee dores y partes:

- Un intento de INSERT para una tupia de envío (VP) tendrá éxito solamente si (a) el número de proveedor de esa tupia existe en V, *y* (b) el número de parte de esa tupia existe como un número de parte en P.
- Un intento de UPDATE para una tupia de envío (VP) tendrá éxito solamente si (a) el número de proveedor en la tupia actualizada existe en V, *y* si (b) el número de parte en la tupia actualizada existe como un número de parte en P.

Observe con detenimiento que las observaciones anteriores se aplican a la varrel *referente,* mientras que las reglas DELETE y UPDATE (explícitas) se aplican a la varrel *referida.* Por lo tanto, hablar acerca de una "regla INSERT", como si esa regla fuera en cierta forma similar a las reglas DELETE y UPDATE existentes, es realmente algo más bien confuso. Este hecho ofrece una justificación adicional para no incluir soporte alguno de la "regla INSERT" en la sintaxis concreta.

8.9

a. Aceptada.
b. Rechazada (viola la unicidad de la clave candidata).

VAF

c. Rechazada (viola la especificación RESTRICT).
d. Aceptada (se elimina el proveedor V3 y todos sus envíos).
e Rechazada (viola la especificación RESTRICT).
f Aceptada (se elimina el proyecto Y4 y todos los envíos del mismo).
g Aceptada.
h Rechazada (viola la unicidad de la clave candidato).
i Rechazada (viola la integridad referencial).
j Aceptada.
k Rechazada (viola la integridad referencial).
1 Rechazada (viola la integridad referencial; el número predeterminado de proyecto en la varrel Y).

8.10 El diagrama referencial se muestra en la figura 8.1. A continuación presentamos ui definición de la base de datos. Por razones de simplicidad, no definimos ninguna restric salvo en la medida que la especificación POSSREP en una definición de tipo dada sirva (j como una restricción *a priori* sobre el tipo.

```
TYPE  CURSO#     POSSREP 1      )
TYPE  TITULO     POSSREP [      )
TYPE  OF#        POSSREP [      )
TYPE  FECHAOF    POSSREP [      )
TYPE  CIUDAD     POSSREP (      )
TYPE  EMP#       POSSREP ;      )
TYPE  NOMBRE     POSSREP [      )
TYPE  PUESTO     POSSREP [      )
TYPE  CALIF      POSSREP [      )
                         i CHAR
VAR   CURSO BASE RELATION
      { CURSO# CURSO#,
        TITULO  TITULO ]
      PRIMARY  KEY { CURSO#

    VAR PRERREQ BASE RELATION
         { CURS0_SUP# CURS0#,
           CURSO_SUB# CURSO# }
         PRIMARY KEY { CURSO_SUP#, CURS0_SUB# }
         FOREIGN KEY { RENAME CURSO_SUP# AS CURSO# }
                                          REFERENCES CURSO
                                          ON DELETE CASCADE
                                          ON UPDATE CASCADE
         FOREIGN KEY { RENAME CURSO_SUB# AS CURSO# }
                                          REFERENCES CURSO
                                          ON DELETE CASCADE
                                          ON UPDATE CASCADE

    VAR OFERTA BASE RELATION { CURSO#   CURS0#, 0F#
          0F#, FECHAOF  FECHAOF, UBICACIÓN CIUDAD }
          PRIMARY KEY { CURSO#, 0F# } FOREIGN KEY {
          CURSO# } REFERENCES CURSO ON DELETE
          CASCADE ON UPDATE CASCADE ;

VAR EMPLEADO  BASE RELATION
    { EMP#    EMP#,
      NOMEMP  NOMBRE,
      PUESTO  PUESTO }
    PRIMARY   KEY { EMP# }
```

■cto ***yyy*** no existe

mos una posible
stricción de tipo,
va (por supuesto)

```
VAR MAESTRO BASE RELATION
     { CURS0# CURSO*, 0F#
     0F#, EMP#  EMP# }
     PRIMARY KEY { CURSO#, 0F#, EMP# } FOREIGN KEY
     { CURSO#, 0F# } REFERENCES OFERTA
                                  ON DELETE CASCADE
                                  ON UPDATE CASCADE
     FOREIGN KEY { EMP# } REFERENCES EMPLEADO
                          ON DELETE CASCADE ON
                          UPDATE CASCADE ;

VAR INCRIPCION BASE RELATION {
     CURSO# CURSO#, 0F#
     OF#, EMP#  EMP#, CALIF
     CALIF }
     PRIMARY KEY { CURSO#, 0F#, EMP# } FOREIGN KEY
     { CURS0#, 0F# } REFERENCES OFERTA
                                  ON DELETE CASCADE
                                  ON UPDATE CASCADE
     FOREIGN KEY { EMP# } REFERENCES EMPLEADO
                          ON DELETE CASCADE ON
                          UPDATE CASCADE :
```

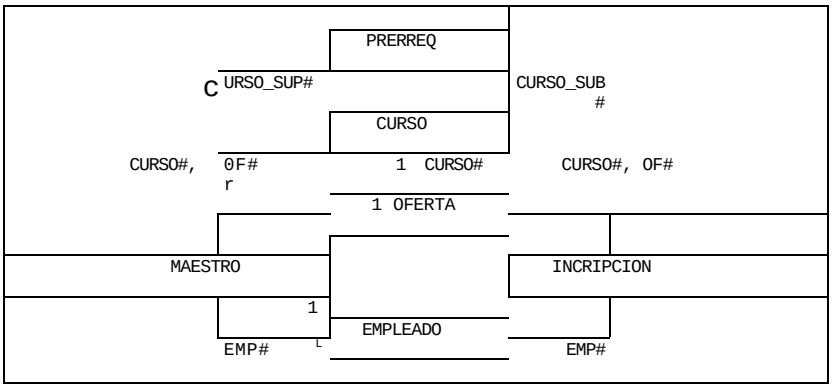


Figura 8.1 Diagrama referencial de la base de datos de educación.

Puntos a destacar:

Los conjuntos de atributos (sencillos) {CURSO*} en MAESTRO y {CURSO*} en INSCRIPCIÓN, también pudieron haber sido considerados como claves externas; ambas haciendo referencia a CURSO. Sin embargo, si las restricciones referenciales de MAESTRO a OFERTA, de INSCRIPCIÓN a OFERTA y de OFERTA a CURSO son mantenidas adecuadamente, las restricciones referenciales de MAESTRO a CURSO y de INSCRIPCIÓN a CURSO serán mantenidas automáticamente. Para una mayor explicación, vea la referencia [8.10].

2. OFERTA es un ejemplo de una varrel que al mismo tiempo es referida y es referente. Existe u restricción referencial hacia OFERTA desde INSCRIPCIÓN (de hecho, también desde TRO) y una restricción referencial de OFERTA hacia CURSO:

INCRIPCION OFERTA CURSO

3. Observe que hay dos rutas referenciales distintas de INSCRIPCIÓN a CURSO: una directa (I clave externa {CURSO#) en **INCRIPCION)** y la otra indirecta a través de OFERTA 11 externas {CURSO#,OF#¡ en INSCRIPCIÓN y {CURSO*} in OFERTA):

INCRIPCION OFERTA T CURSO

Sin embargo, las dos rutas en realidad no son independientes entre sí (la ruta superior está in cita por la combinación de las dos inferiores). Para una mayor explicación de este punto, consv una vez más la referencia [8.10].

4. También existen dos rutas referenciales distintas de PRERREQ a CURSO, pero esta vez las de rutas en realidad *son* independientes (tienen significados totalmente independientes). Vea o vez la referencia [8.10].

8.11 El diagrama referencial es mostrado en la figura 8.2. Observe que la base de datos comprende ciclo referencial (hay una ruta referencial desde cada una de las dos varrels hacia sí mismas). Adem de esta consideración, la definición de la base de datos es básicamente directa. Omitimos los detalles.

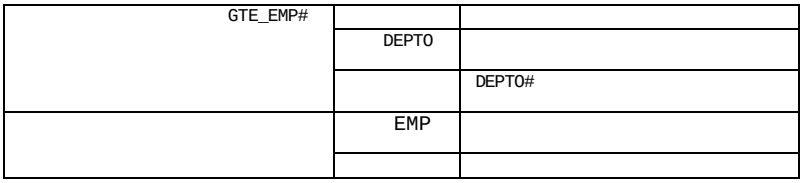


Figura 8.2 Diagrama referencial que comprende un ciclo. **8.12**

Mostramos solamente las definiciones de varrels (sólo en bosquejo):

```
VAR EMP BASE RELATION
      { EMP#   . . . ,
          ________
         PUESTO ... } PRIMARY
    KEY { EMP# } ;

VAR PGMR BASE RELATION
      { EMP#  ... ,
          _______
    LENG  . . . } PRIMARY KEY { EMP# }
    FOREIGN KEY { EMP# } REFERENCES EMP ON
    DELETE CASCADE ON UPDATE CASCADE ;
```

Puntos a destacar:

1. Este ejemplo ilustra la idea de que una clave externa también puede ser una clave candidata de la varrel que la contiene. La varrel EMP contiene todos los empleados y la varrel PGMR

contiene sólo aquellos empleados que son programadores; por lo tanto, todo número de empleado que aparezca en PGMR debe también aparecer en EMP (pero lo opuesto no es cierto). La clave primaria de PGMR es también una clave externa, que hace referencia a la clave primaria de EMP.

2. Observe que en este ejemplo hay otra restricción que necesita ser mantenida; es decir, la restricción de que un empleado dado aparecerá en PGMR si y solamente si el valor de PUESTO de ese empleado es "Programador". Por supuesto, esta restricción no es una restricción *referential*.

8.14 Observe que las soluciones a. y b. siguientes sólo son aproximaciones a sus contrapartes mostradas en la respuesta al ejercicio 8.1.

a.
```
CREATE DOMAIN CIUDAD CHAR(15) VARYING
    CONSTRAINT CIUDADES_VALIDAS
        CHECK ( VALUE IN ( 'Londres', 'Paris', 'Roma',
                           'Atenas',   'Oslo',   'Estocolmo' ,
                           'Madrid', 'Amsterdam' ) ) ;
```

b.
```
CREATE DOMAIN V# CHAR(5) VARYING
CONSTRAINT V#_VALID0 CHECK
       ( SUBSTRING ( VALUE FROM 1 FOR 1 ) = 'V' AND CAST (
  SUBSTRING ( VALUE FROM 2 ) AS INTEGER ) >= 0 AND CAST (
  SUBSTRING ( VALUE FROM 2 ) AS INTEGER ) <= 9999 ) ;
```

C.
```
CREATE ASSERTION SQLC CHECK
            ( P.COLOR <> 'Rojo' OR P.PESO < 50.0 ) ;
```

d.
```
CREATE ASSERTION SQL_D CHECK
            ( NOT EXISTS ( SELECT * FROM Y YX WHERE
                  EXISTS ( SELECT * FROM Y YY WHERE (
                  YX.Y# <> YY.Y# AND
                           YX.CIUDAD = YY.CIUDAD ) ) ) ) ;
```

e.
```
CREATE ASSERTION SOLE CHECK
            ( ( SELECT C0UNT(*) FROM V
                WHERE V.CIUDAD - 'Atenas' ) < 2 ) ;
```

f.
```
CREATE ASSERTION SQL_F CHECK
            ( NOT EXISTS ( SELECT *
                           FROM  VPY VPYX
                           WHERE  VPYX.CANT > 2 *
                                ( SELECT AVG ( VPYY.CANT )
                                  FROM  VPY VPYY ) ) ) ;
```

g.
```
CREATE ASSERTION SQL_G CHECK
            ( NOT EXISTS ( SELECT * FROM V VX WHERE
                  EXISTS ( SELECT * FROM V VY WHERE
                           VX. STATUS = ( SELECT MAX ( V. STATUS )
                                          FROM V ) AND VY.STATUS
                           = ( SELECT MIN ( V.STATUS )
                                          FROM V ) AND
                           VX.STATUS <> VY.STATUS AND
                           VX.CIUDAD = VY.CIUDAD ) ) ) ;
```

h.
```
CREATE ASSERTION SQL_H CHECK
            ( NOT EXISTS ( SELECT * FROM Y WHERE
              NOT EXISTS ( SELECT * FROM V WHERE
                           V.CIUDAD = Y.CIUDAD ) ) ) ;
```

```
i. CREATE ASSERTION SQL_I CHECK
             ( NOT EXISTS ( SELECT * FROM Y WHERE
               NOT EXISTS ( SELECT * FROM V WHERE
                            V.CIUDAD = Y.CIUDAD AND
                            EXISTS ( SELECT * FROM VPY
                                     WHERE VPY.V# = V.V#
                                      AND   VPY.Y# = Y.Y# ) ) ) )

   CREATE ASSERTION SQL_J CHECK
             ( NOT EXISTS ( SELECT * FROM P
                OR EXISTS ( SELECT * FROM P
                            WHERE   P.COLOR    'RO]O' )
```

k.

```
CREATE ASSERTION SQLK CHECK
            ( ( SELECT AVG ( V. STATUS    FROM V ) > 10
```

Si la tabla de proveedores está vacía, el operador AVG de SQL regresará (¡en forma incorrecta!) un valor nulo; la expresión condicional dará como resultado *desconocida* y la restricción no se verá como violada. Para una mayor explicación, consulte el capítulo 18.

```
   CREATE ASSERTION SQLL CHECK
             ( NOT EXISTS ( SELECT  * FROM V
                            WHERE  V.CIUDAD ■ 'Londres'
                            AND    NOT EXISTS
                                 ( SELECT * FROM VPY
                                   WHERE  VPY.V# = V.V#
                                   AND    VPY.P# = 'P2'
```

```
m. CREATE ASSERTION SQLM CHECK
             ( NOT EXISTS ( SELECT
                            WHERE              Rojo' )
                OR EXISTS ( SELECT
                            WHERE              Rojo'
                            AND
                                   * FROM P
                                   P.COLOR =
                                   * FROM P
                                   P.COLOR •
                                   P.PESO < 50.0 ) )
```

```
n. CREATE ASSERTION SQLN CHECK
             ( ( SELECT COUNT(* ) FROM P
             WHERE EXISTS       ( SELECT * FROM VPY WHERE
             EXISTS             ( SELECT * FROM V WHERE (
                                P.P# = VPY.P# AND
                                  VPY.V# = V.V# AND
                                 V.CIUDAD = 'Londres' ) ) ) ) >
                 SELECT COUNT(*) FROM P
                 WHERE EXISTS    ( SELECT * FROM VPY WHERE
                 EXISTS          ( SELECT * FROM V WHERE
                                ( P.P# = VPY.P# AND
                                  VPY.V# = V.V# AND
                                  V.CIUDAD - 'Paris' ) ) ) ) ;
```

```
O. CREATE ASSERTION SQLO CHECK
             ( ( SELECT SUM ( VPY. CANT ) FROM VPY
                 WHERE  ( SELECT V.CIUDAD FROM V
                          WHERE V.V# = VPY.V# ) ■ 'Londres' ) > (
               SELECT SUM ( VPY.CANT ) FROM VPY WHERE  ( SELECT
               V.CIUDAD FROM V
                            WHERE V.V# = VPY.V# ) ■ 'Paris' ) ) ;
```

p. No puede hacerse,

q. No puede hacerse.

CAPITULO 9

Vistas

9.1 INTRODUCCIÓN

cta!)
w se

Como explicamos en el capítulo 3, una **vista** es básicamente sólo una expresión del álgebra relacional (o de algo equivalente al álgebra relacional) con un nombre. Por ejemplo:

```
VAR BUEN_PROVEEDOR VIEW
  ( V WHERE STATUS > 15 ) { V#, STATUS, CIUDAD } ;
```

Al ejecutar esta instrucción, la expresión del álgebra relacional (es decir, **la expresión que define la vista)** no es evaluada sino que simplemente es "recordada" por el sistema, el cual de hecho la guarda en el catálogo bajo el nombre especificado BUEN_PROVEEDOR. Sin embargo, para el usuario ahora es como si en realidad existiera en la base de datos una varrel denominada BUEN_PROVEEDOR, con las tupias y atributos que muestra la figura 9.1 en las partes no sombreadas (por supuesto, tomamos nuestros valores de datos usuales). En otras palabras, el nombre BUEN_PROVEEDOR denota una varrel **derivada (y virtual),** cuyo valor en cualquier momento es la relación que resultaría si la expresión que define la vista se evaluara en ese momento.

En el capítulo 3, también explicamos que una vista como BUEN_PROVEEDOR es en efecto sólo una *ventana* para los datos subyacentes. Cualquier actualización a esos datos subyacentes será visible automática e inmediatamente a través de esa ventana (por supuesto, siempre que estén dentro del alcance de la vista); de manera similar, cualquier actualización a la vista será aplicada automática e inmediatamente a los datos subyacentes y por lo tanto, serán visibles a través de la ventana.

Ahora bien, dependiendo de las circunstancias, el usuario podría o no darse cuenta de que BUEN_PROVEEDOR es en realidad una vista. Algunos usuarios podrían estar al tanto de

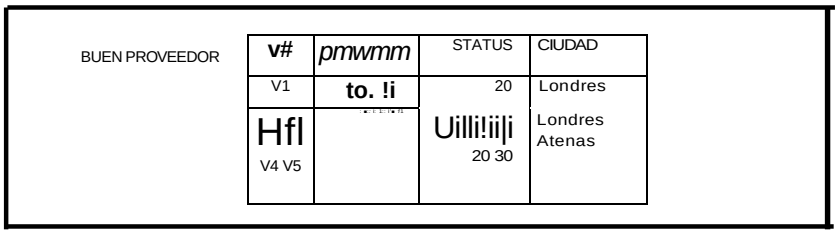


Figura 9.1 BUENLPROVEEDOR como una vista de la varrel base V (partes sin sombrear).

este hecho y podrían entender que bajo ella existe una varrel "real" V; otros podrían creer gi nuinamente que BUEN_PROVEEDOR es una varrel "real" por derecho propio. De cualqu manera, no hay mucha diferencia; la idea es que los usuarios pueden operar sobre BUEN_PI VEEDOR tal como si *fuese* una varrel real. Por ejemplo, aquí tenemos una consulta sobre BUEN_ PROVEEDOR:

```
BUEN_PROVEEDOR WHERE CIUDAD * 'Londres'
```

Dados los datos de ejemplo de la figura 9.1, el resultado es:

V#	STATUS	CIUDAD
V3	30	París
V5	30	Atenas

Esta consulta luce ciertamente como una consulta normal sobre una varrel "real". Y como vimos en el capítulo 3, el sistema maneja dicha consulta convirtiéndola en una consulta *t* valente sobre la variable (o variables, en plural) de relación base subyacente. Esto lo hacera plazando efectivamente cada aparición del *nombre* de la vista en la consulta por la expresión q *define* la vista. En el ejemplo, este **procedimiento de sustitución** nos da

```
( ( V WHERE STATUS > 15 ) { V#, STATUS, CIUDAD } )
                              WHERE CIUDAD * 'Londres'
```

lo cual se puede ver como equivalente a la forma más sencilla

```
( V WHERE STATUS > 15 AND CIUDAD * 'Londres' )
                                    { V#, STATUS, CIUDAD }
```

Y esta consulta produce el resultado que mostramos antes.

Por cierto, vale la pena señalar que el proceso de sustitución que acabamos de describ decir, el proceso de sustituir la expresión que define la vista por el nombre de la vista)*funcioia precisamente debido a la propiedad de cierre relacional.* Entre otras cosas, el cierre implica q siempre que el nombre de una varrel simple *R* pueda aparecer dentro de una expresión, en lugar podrá aparecer una expresión relacional de complejidad arbitraria (siempre y cuando dé como resultado una relación del mismo tipo que *R).* En otras palabras, las vistas funcionan pi cisamente debido que las relaciones son cerradas bajo el álgebra relacional; un ejemplo m la importancia fundamental de la propiedad de cierre.

Las operaciones de actualización se tratan de manera similar. Por ejemplo, la operación

```
UPDATE BUEN_PROVEEDOR WHERE CIUDAD = 'París'
       STATUS := STATUS + 10 ;
```

se convierte efectivamente en

```
UPDATE V WHERE STATUS >  15 AND CIUDAD = 'París'
       STATUS := STATUS  + 10 ;
```

Las operaciones INSERT y DELETE se manejan de manera similar.

Irían creer ge-
De cualquier
BUEN_PRO-
ansulta sobre

Más ejemplos

En esta subsección presentamos varios ejemplos, para referencia posterior.

```
1. VAR PARTEROJA VIEW
       ( ( P WHERE COLOR = COLOR (  'Rojo' ) ) {ALL BUT COLOR } )
                                          RENAME PESO AS PS ;
```

La vista PARTEROJA tiene los atributos P#, PARTE, PS y CIUDAD, y contiene sólo tupias de partes rojas.

```
2. VAR CP VIEW
       SUMMARIZE VP PER P { P# } ADD SUM ( CANT ) AS CANTOT ;
```

A diferencia de BUEN_PROVEEDOR y PARTEROJA, la vista CP no es un simple subconjunto (es decir una restricción o proyección) de cierta varrel subyacente. En su lugar puede ser considerada como una especie de *resumen estadístico* o *compresión* de esa varrel subyacente.

al". Y como
nsulta equi-
hace reem-
presión que

```
3. VAR DOSCIUDADES VIEW
       ( ( V RENAME CIUDAD AS CIUDADV ) JOIN VP JOIN
         ( P RENAME CIUDAD AS CIUDADP ) ) { CIUDADV, CIUDADP } ;
```

En general, en la vista DOS_CIUDADES aparece un par de nombres de ciudad *(x,y)* si y solamente si un proveedor ubicado en la ciudad *x* suministra partes almacenadas en la ciudad *y*. Por ejemplo, el proveedor VI suministra la parte Pl; el proveedor VI está ubicado en Londres y la parte Pl está almacenada en Londres, de modo que en la vista aparece el par (Londres, Londres).

```
4. VAR PARTEROJA_PESADA VIEW
       PARTEROJA WHERE PS > PESO    12.0
```

Este ejemplo muestra una vista definida en términos de otra.

scribir (es
) funciona
íplica que
¡ón, en su
uando dé
onan pre-
o más de

ración

Definición y eliminación de vistas

Aquí tenemos entonces la sintaxis para definir una vista:

```
VAR <nombre de varrel> VIEW <expresión relacional>
                    <lista de definición de claves candidatas> ;
```

Se permite que la <*lista de definición de claves candidatas*> esté vacía, debido a que el sistema debe tener la posibilidad de *inferir* las claves candidatas de las vistas [10.6]. Por ejemplo, en el caso de BUEN_PRO VEEDOR, el sistema debe estar al tanto de que la única clave candidata es {V#}, heredada de la varrel subyacente V.

Observamos que para usar la terminología ANSI/SPARC del capítulo 2, Jas definiciones de vistas combinan (a) la función de *esquema externo* y (b) la función de *transformación externa/conceptual.* Esto se debe a que especifican tanto (a) cómo luce el objeto externo (es decir, la vista) como (b) la manera en que ese objeto se asocia con el nivel conceptual (es decir, con las varrels base subyacentes). *Nota:* Algunas definiciones de vistas no especifican la transformación externa/conceptual como tal, sino más bien una transformación *externa/externa.* La vista PARTEROJA PESADA de la subsección anterior es uno de estos casos.

La sintaxis para eliminar una vista es

```
DROP VAR <nombre de varrel> ;
```

donde, por supuesto, el *Knombre de varrel>* se refiere específicamente a una vista. Ahora bie en el capítulo 5 explicamos que un intento por eliminar una varrel base fracasaría si cualquii vista se refería en ese momento a dicha varrel base. De manera similar, expusimos que un in tentó por eliminar una vista también fracasaría si alguna otra definición de vista se refiriera (ese momento a dicha vista. En forma alternativa (y por analogía con las restricciones refen ciales), podríamos considerar la extensión de la instrucción de definición de vista para incluir alguna clase de opción de "RESTRICT vi. CASCADE"; RESTRICT (la opción predeterminada) significaría que debería fracasar un intento por eliminar cualquier varrel referida en la definición de la vista. CASCADE significaría que dicho intento debería tener éxito y actuaría "en cascada" para eliminar también la vista que hace la referencia. *Nota:* SQL sí soporta dicha opción, pero la pone en la instrucción DROP en lugar de la definición de la vista. No existe opción predeterminada, si se requiere de una, ésta deberá ser declarada en forma explícita (vea la sección 9.6).

9.2 ¿PARA QUÉ SON LAS VISTAS?

Existen muchas razones por las cuales es necesario el soporte de vistas. Aquí tenemos algunas:

- *Las vistas proporcionan seguridad automática para datos ocultos*

  "Datos ocultos" se refiere a los datos que no son visibles a través de alguna vista determi nada (por ejemplo, los nombres de proveedor en el caso de la vista BUEN_PROVEED0I Existe la seguridad de que estos datos no serán accedidos (por lo menos, en un acceso de recuperación) a través de esa vista en particular. Por lo tanto, obligar a los usuarios a acceder a la base de datos a través de vistas constituye un mecanismo de seguridad simple pero efectivo. En el capítulo 16 hablaremos más sobre este uso particular de las vistas.

- *Las vistas ofrecen una posibilidad de forma abreviada o "macro"*

  Considere la consulta "Obtener las ciudades que almacenan partes que están disponibles d< algún proveedor en Londres". Dada la vista DOS_CIUDADES de la subsección "Más ejemplos" anterior, la siguiente formulación satisface el requerimiento:

  ```
  ( DOSCIUDADES WHERE CIUDADV = 'Londres' ) { CIUDADP }
  ```

  En contraste, sin la vista la consulta es mucho más compleja:

  ```
  ( ( ( V RENAME CIUDAD AS CIUDADV ) JOIN VP
        JOIN ( P RENAME CIUDAD AS CIUDADP ) )
                  WHERE CIUDADV = 'Londres' ) { CIUDADP }
  ```

  Aunque el usuario *podría* usar directamente esta segunda formulación —por supuesto si las restricciones de seguridad lo permiten—, la primera es obviamente más simple. (Desde luego, la primera es en realidad sólo una forma abreviada de la segunda; el mecanismo de procesamiento de vistas del sistema expandirá efectivamente la primera form! lación en la segunda antes de que ésta sea ejecutada.)

Existe una fuerte analogía con las **macros** de un sistema de lenguaje de programación. En principio, un usuario de un sistema de lenguaje de programación *podría* escribir directamente la forma extendida de una macro determinada en su código fuente, pero es mucho más conveniente (por varias razones bien entendidas) no hacerlo así, sino más bien usar la forma abreviada de la macro y dejar que el procesador de macros del sistema realice la expansión en beneficio del usuario. Observaciones similares se aplican a las vistas. Por lo tanto, las vistas en un sistema de base de datos juegan un papel en cierta forma similar al de las macros en un sistema de lenguaje de programación; y las bien conocidas ventajas y beneficios de las macros también se aplican, *haciendo los cambios necesarios,* directamente a las vistas. Observe en particular que (al igual que el caso de las macros) el uso de las vistas no implica ninguna sobrecarga adiciona] del rendimiento en tiempo de ejecución; sólo hay una pequeña sobrecarga en el tiempo de procesamiento de la vista (similar al tiempo de expansión de una macro).

- *Las vistas permiten que los datos sean vistos de distinta forma por diferentes usuarios al mismo tiempo*

  En efecto, las vistas permiten a los usuarios concentrarse solamente en la porción de la base de datos que les interesa e ignorar el resto (y quizás reestructurarla de manera lógica). Ésta es obviamente una consideración importante cuando hay muchos usuarios distintos, con muchos requerimientos diferentes, interactuando al mismo tiempo con una sola base de datos integrada.

- *Las vistas pueden ofrecer la independencia lógica de los datos*

  Esta es una de las ideas más importantes. Vea la sección siguiente.

Independencia lógica de los datos

Le recordamos que la independencia lógica de los datos puede definirse como *la inmunidad de los usuarios y los programas de usuario ante los cambios en la estructura lógica de la base de datos* (donde por *estructura lógica* entendemos el nivel conceptual o "lógico de la comunidad"; vea el capítulo 2). Y por supuesto, las vistas son el medio por el cual se logra Ja independencia lógica de los datos en un sistema relacional. Existen dos aspectos relacionados con esa independencia lógica de los datos y son el **crecimiento y** la **reestructuración.** *Nota:* Exponemos aquí el *crecimiento* principalmente por cubrir el tema completo; aunque es importante, tiene poco que ver con las vistas como tales.

- *Crecimiento*

  Conforme la base de datos crece para incorporar nuevas clases de información, la definición de la misma debe crecer de igual forma. Hay dos posibles clases de crecimiento que pueden ocurrir;

  1. La expansión de una varrel base existente para que incluya un nuevo atributo, lo cual corresponde a la incorporación de nueva información relativa a cierto tipo de objeto exis tente; por ejemplo, agregar un atributo DESCUENTO a la varrel base de proveedores.
  2. La inclusión de una nueva varrel base, lo que corresponde a la incorporación de un nuevo tipo de objeto; por ejemplo, agregar información de proyectos a la base de datos de proveedores y partes.

  Ninguno de estos cambios üebe tener efecto alguno sobre los usuarios o programas de usuario existentes; por lo menos en principio (pero vea el ejemplo 7.7.1 del capítulo 7, el cual contiene una observación específica con respecto a SQL).

Reestructuración

Ocasionalmente podría ser necesario reestructurar la base de datos de una manera t aunque el contenido de la información general permanezca igual, la *ubicación logic* información sea modificada; es decir, que se altere de alguna manera la asignación de butos para las varrels base. Consideramos aquí solamente un ejemplo sencillo. Supon¡ por alguna razón (para los fines del presente libro no es importante precisarla reemplazar la varrel base V por las dos varrels base siguientes:

```
VAR VNC BASE RELATION { V# V#, PROVEEDOR NOMBRE, CIUDAD CHAR }
    PRIMARY KEY { V# } ;

VAR VT BASE RELATION { V# V#, STATUS INTEGER }
    PRIMARY KEY { V# } ;
```

En este punto, la idea crucial que debemos observar es que la *antigua varrel Vt junta de las dos nuevas varrels VNC y VT* (además de que VNC y VT son *proyectil* esa varrel anterior V). Así que creamos una vista que es exactamente esa junta y la dent namos V:

```
VAR V VIEW
    VNC JOIN VT ;
```

Todo programa de aplicación u operación interactiva que anteriormente se refería varrel base V, en su lugar se referirá ahora a la vista V. Por lo tanto, *siempre que el* sislf *soporte correctamente operaciones de manipulación de datos sobre vistas,* los usuaric programas de usuario serán inmunes de manera lógica a esta reestructuración particular (la base de datos.*

Como un punto aparte, observamos que la sustitución de la varrel original de proveedores V por sus dos proyecciones VNC y VT, no es en lo absoluto un asunto trivial. Enp ticular observe que debe hacerse algo con la varrel de envío VP, y a que esa varrel tiene i clave externa que hace referencia a la varrel de proveedores original. Vea el ejercicio 9.1 al final del capítulo.

Para retomar el hilo de la explicación principal: está claro que del ejemplo VNC-V no podemos deducir que la independencia lógica de los datos pueda lograrse de cara a *toe las reestructuraciones posibles.* El aspecto crítico es si existe una transformación sin bigüedad de la versión reestructurada de la base de datos al regresar a la versión anteri (es decir, si la reestructuración es reversible); o en otras palabras, si las dos versiones **equivalentes en información.** Si no lo son, resulta claro que no se puede lograr la ¡ni pendencia lógica de los datos.

9.3

* ¡ En principio! Por desgracia, la mayoría de los productos SQL actuales (y el estándar de SQL) *no* sop tan correctamente las operaciones de manipulación de datos sobre vistas y por lo tanto, no ofrecen el | necesario de inmunidad a los cambios que ofrecen los productos del ejemplo. Para ser más específicos.i gunos productos (no todos) sí soportan correctamente las recuperaciones de vistas, pero ningún prodi —hasta donde yo sé— soporta actualizaciones de vistas ! 00 por ciento correctas. Por lo tanto, algunos pn ductos sí proporcionan una total independencia lógica de los datos para las operaciones de recuperacii pero actualmente ninguno lo hace para las operaciones de actualización.

Dos principios importantes

La explicación anterior sobre la independencia lógica de los datos hacer surgir otra idea importante. El hecho es que las vistas sirven a dos propósitos un tanto diferentes:

- Obviamente, un usuario que de hecho *define* una vista *V,* está consciente de la expresión de definición de la vista *X* correspondiente; ese usuario puede emplear el nombre *V* siempre que tenga como intención la expresión *X,* pero (como ya vimos) estos usos son básicamente una forma abreviada.
- Por otra parte, un usuario que sólo está enterado de que existe esa vista *V* y que está dis ponible para su uso, *no* está generalmente al tanto de la expresión *X* que define la vista; de hecho, la vista *V* debe verse y comportarse exactamente como una varrel base.

Como resultado de lo anterior, subrayamos ahora la idea de que la cuestión de cuáles varrels son base y cuáles son derivadas (es decir, vistas) es en gran medida arbitraria. Por ejemplo, considere el caso de las varrels V, VNC y VT de la "reestructuración" expuesta en la subsección anterior. Debe quedar claro que podríamos (a) definir V como la varrel base y VNC y VT como vistas de proyecciones de la primera; *o bien* (b) definir VNC y VT como varrels base y V como una vista de junta para ambas.* *De esto se deduce que no debe haber distinciones arbitrarias e innecesarias entre las varrels base y las derivadas.* Nos referimos a este hecho como ***El principio de intercambiabilidad*** (de varrels base y derivadas). En particular, observe que este principio implica que *debemos* tener la posibilidad de actualizar las vistas; la capacidad de actualización de la base de datos no debe depender de nuestra decisión (esencialmente arbitraria) sobre cuáles varrels decidimos que sean las base y cuáles las vistas. Para una mayor explicación, vea la sección 9.4.

Pongámonos de acuerdo momentáneamente para referirnos al conjunto de todas las varrels base como la base de datos "real". Pero (en general) un usuario típico interactúa no con la *propia* base de datos real, sino con lo que podríamos llamar una base de datos "expresable", la cual (en general) consiste en alguna combinación de varrels base y vistas. Ahora bien, podemos dar por hecho que ninguna de las varrels de esa base de datos expresable puede ser derivada de las demás (ya que dicha varrel podría ser eliminada sin pérdida de información). Por lo tanto, *desde el punto de vista del usuario,* dichas varrels son todas (¡por definición!) varrels base, y en realidad son independientes entre sí (es decir, para usar la terminología del capítulo 3: todas son autónomas). Y de forma similar para la propia base de datos; es decir, la elección de qué base de datos es la "real" es una elección arbitraria (mientras todas las elecciones tengan información equivalente). Nos referimos a este hecho como ***El principio de la relatividad de la base de datos.***

9.3 RECUPERACIÓN DE VISTAS

Ya explicamos en general cómo una operación de recuperación sobre una vista se convierte en una operación equivalente sobre las varrels subyacentes. Ahora haremos nuestra explicación un poco más formal, como sigue:

---

*Vea la explicación de la *descomposición sin pérdida* en el capítulo 11, sección 11.2.

'En este punto, ignoramos todas las vistas definidas por el usuario en cuestión; las cuales (como ya vimos) son solamente formas abreviadas.

Primero, observe que cualquier expresión relacional dada puede ser considerada como una **función** con valor de relación; dados los valores de las distintas varrels mencionadas en la expresión (que representan los argumentos para esta invocación particular de la función), la expresión produce otra relación. Ahora bien, sea *D* una base de datos (la cual, para los fines actuales. veremos sólo como un conjunto de varrels) y sea V una vista sobre *D* (es decir, una vista cuya expresión definitoria X es alguna función sobre *D):*

```
V = X ( D )
```

Sea *R* una operación de recuperación sobre *V;* por supuesto, *R* es otra función con valor de relación y el resultado de la recuperación es

```
f l ( V ) = f l ( X ( D ) )
```

Por lo tanto, el resultado de la recuperación se define como igual al resultado de aplicar X a D; es decir, de **materializar** una copia de la relación que es el valor actual de la vista Vy luego aplicar *R* a esa copia materializada. Ahora bien, en la práctica es ciertamente más eficiente usar en su lugar el procedimiento de **sustitución,** como explicamos en la sección 9.1; y ahora podemos ver que ese procedimiento es equivalente a formar la función C que es la *composición R(X)* de las funciones X y *R,* en ese orden, y luego aplicar directamente C a *D.* Sin embargoes conveniente (por lo menos al nivel conceptual) definir la semántica de la recuperación de la vista en términos de la materialización en lugar de la sustitución; en otras palabras, la sustituciones válida siempre y cuando se garantice que produzca el mismo resultado que produciría si en su lugar se empleara la materialización (y por supuesto, así *está* garantizado).

Ahora, por nuestras exposiciones previas, usted ya debe estar familiarizado básicamente con la explicación anterior. Aquí lo hacemos explícito por las siguientes razones:

- Primero, establece la base para una explicación similar (aunque más de búsqueda) de las operaciones de actualización de la sección 9.4 siguiente.
- Segundo, pone en claro que la materialización es una técnica perfectamente legítima de im plementation de vistas (aunque es probable que sea más bien ineficiente), por lo menos para operaciones de recuperación. Pero por supuesto, no puede ser usada para operaciones de ac tualización, ya que toda la idea de actualizar una vista consiste precisamente en aplicarlas actualizaciones a las varrels base subyacentes (no sólo a cierta *copia* materializada de los datos). Una vez más, vea adelante la sección 9.4.
- Tercero, aunque el procedimiento de sustitución es bastante directo y en teoría funciona per fectamente bien en el 100 por ciento de los casos, el hecho triste es que (al momento déla publicación de este libro) existen algunos productos de SQL para los cuales éste *no* funcio na en la práctica!; es decir, hay algunos productos de SQL en los cuales algunas recupera ciones sobre ciertas vistas fallan de manera sorprendente. En la práctica, el procedimiento tampoco funciona en versiones del SQL estándar anteriores al SQL/92. Y la razón de las fallas es precisamente que los productos en cuestión y las versiones anteriores del estándar de SQL, no soportan completamente la propiedad relacional de cierre. Vea la parte a. del ejercicio 9.14, al final del capítulo.

9.4 ACTUALIZACIÓN DE VISTAS

El problema de la actualización de vistas puede expresarse entonces como: dada una actualización en particular sobre una vista determinada, ¿qué actualizaciones necesitan ser aplicadas a qué varrels subyacentes para poder implementar la actualización de la vista original? Para ser más precisos, sea *D* una base de datos y sea *V* una vista sobre *D;* es decir, una vista cuya definición *X* es una función sobre *D:*

```
V = X ( D )
```

(como en la sección 9.3). Ahora, sea *U* una operación de actualización sobre *V. U* puede ser considerada como una operación que tiene el efecto de cambiar el argumento; lo que produce

```
a ( v ) - v ( x ( o ) )
```

Entonces, el problema de actualizar la vista es el mismo que encontrar una operación de actualización *U'* sobre *D* tal que

```
U ( X ( D ) ) = X ( U ' ( D ) )
```

ya que (por supuesto) *D* es lo único que "en realidad existe" (las vistas son virtuales); así que las actualizaciones no pueden ser implementadas directamente en términos de vistas *como tales.* Antes de continuar, debemos enfatizar que el problema de la actualización de vistas ha sido tema de considerable investigación durante años, y que han sido propuestos muchos enfoques diferentes para su solución (por mi parte y la de otros autores); por ejemplo, vea las referencias [9.7], [9.10-9.13], [9.15] y en particular las propuestas de Codd para RM/V2 [5.2]. En este capítulo describimos un enfoque comparativamente nuevo [9.9], el cual es menos adecuado que las propuestas anteriores, pero tiene la virtud de ser compatible a futuro con los mejores aspectos de dichas propuestas. También tiene la virtud de considerar como actualizables a un rango de vistas mucho más amplio que el de los enfoques previos; de hecho, trata a *todas* las vistas como potencialmente actualizables, salvo que violen las restricciones de integridad.

Nueva visita a la regla de oro

Recordemos *La regla de oro* del capítulo anterior:

> *Nunca debe permitirse que una operación de actualización deje a cualquier varrel en un estado que viole su propio predicado.*

Cuando presentamos por primera vez esta regla, hicimos énfasis en la idea de que se aplica a *todas* las varrels (lo mismo derivadas que base). En otras palabras, las varrels derivadas también tienen predicados —como de hecho deben tener, en virtud del *principio de intercambiabilidad*— y el sistema necesita saber cuáles son esos predicados para realizar la actualización de vistas en forma correcta. Pero ¿cómo luce el predicado de una vista? Resulta claro que lo que necesitamos es un conjunto de **reglas de inferencia de predicados,** de manera que si conocemos los predicados de las entradas para cualquier operación relacional, podemos inferir el predicado de salida de esa operación. Dado dicho conjunto de reglas, podemos inferir el predicado de una vista a partir de los predicados de las varrels base en cuyos términos está definida la vista,

directa o indirectamente. (Desde luego, ya conocemos los predicados de esas varrels base: son el AND lógico de cualesquiera que sean las restricciones de la varrel declaradas para la varrel en cuestión; por ejemplo, restricciones de clave candidata).

De hecho, es muy fácil encontrar dicho conjunto de reglas; están inmediatamente después de las definiciones de los operadores relaciónales. Por ejemplo, si *A* y *B* son dos varrels cualesquiera del mismo tipo, si sus respectivos predicados son *PA* y *PB,* y si se define la vista *C* como *A* INTERSECT *B,* entonces el predicado *PC* de esa vista es obviamente *(PA)* AND *(PB);* es decir, una tupia determinada aparecerá en *C* si, y solamente si, *PA* y *PB* son *verdaderos.* Más adelante consideraremos otros operadores relaciónales.

Nota: Por lo tanto, las varrels derivadas "heredan" automáticamente ciertas restricciones de las varrels de las que se derivan. Aunque es posible que una varrel dada esté sujeta a ciertas restricciones adicionales por encima y arriba de las heredadas. De ahí que sea necesario poder declarar explícitamente las restricciones para las varrels derivadas (un ejemplo podría ser una definición de clave candidata para una vista); de hecho, **Tutorial D** soporta esta posibilidad. Sin embargo, por razones de simplicidad, a partir de este punto ignoraremos casi por completo esta posibilidad.

Hacia un mecanismo de actualización de vistas

Existen varios principios importantes que todo enfoque semántico debe satisfacer respecto del problema de actualización de vistas (por supuesto, *la regla de oro* es el principal, aunque no el único). Los principios en cuestión son los siguientes:

1. La posibilidad de actualización de vistas es un aspecto semántico, no sintáctico; es decir, no debe depender de la forma sintáctica en particular en la que esté declarada la vista en cuestión. Por ejemplo, las siguientes dos definiciones son semánticamente idénticas:

   ```
   VAR V1 VIEW
       V WHERE STATUS > 25 OR CIUDAD = 'París' ;

   VAR V1 VIEW
     ( V WHERE STATUS > 25 ) UNION ( V WHERE CIUDAD = 'París' ) ;
   ```

   De manera obvia, ambas vistas deben ser actualizables o no serlo (de hecho, es obvio que ambas deben ser actualizables). En contraste, el estándar de SQL, y la mayoría de los productos SQL actuales, adoptan la posición *ad hoc* de que la primera es actualizable y la segunda no lo es (vea la sección 9.6).

2. Del punto anterior podemos deducir que el mecanismo de actualización de vistas debe fun cionar correctamente en el caso especial en que la "vista" sea de hecho una varrel base; esto se debe a que toda varrel base *B* no es distinguible semánticamente de una vista *V* definida como *B* UNION *B o B* INTERSECT Bofi WHERE *verdadera,* o cualquiera de muchas otras expresiones que son equivalentes de manera idéntica a *B.* Por lo tanto, las reglas para actualizar (por ejemplo) una vista de unión cuando se aplican a la vista *V = B* UNION *B,* deben producir exactamente el mismo resultado que si la actualización en cuestión se aplicara directamente a la varrel base *B.* En otras palabras, el tema de esta sección aunque la anunciamos como "actualización de vistas", en realidad es "actualización de *varrels"* en general; describiremos una teoría que funcione para actualizar *todas* las varrels, no sólo para las vistas.

3. Donde corresponda, las reglas de actualización deben preservar la simetría. Por ejemplo, la regla de DELETE para una vista de intersección *V* = *A* INTERSECT *B*, no debe provocar que una tupia se elimine arbitrariamente de A y no de *B*, aunque dicha eliminación de un solo lado ciertamente tendría el efecto de eliminar una tupia de la vista. En su lugar, la tupia debe ser eliminada tanto de *A* como de *B*. (En otras palabras, *no debe existir ambigüedad;* siempre debe haber una sola manera de implementar una determinada actualización: una forma que funcione en todos los casos. En particular, no debe existir una diferencia lógica entre una vista definida como *A* INTERSECT *B* y otra definida como *B* INTERSECT A.)
4. Las reglas de actualización deben tomar en cuenta cualquier procedimiento disparado apli cable, incluyendo en particular acciones referenciales como un DELETE en cascada.
5. Por razones de simplicidad, entre otras razones, es necesario considerar a UPDATE como una forma abreviada de una secuencia DELETE-INSERT, y así lo consideraremos. Esta forma abreviada es aceptable *siempre y cuando* sea entendido que:
   - No se realizan verificaciones de predicados de varrels "a la mitad de" cualquier actualiza ción dada; es decir, la expansión de UPDATE es DELETE-INSERT-verificación, en vez de DELETE-verificación-INSERT-verificación. Por supuesto, la razón es que el DELETE podría violar temporalmente el predicado de la varrel mientras que el UPDATE en con junto no; por ejemplo, suponga que la varrel *R* contiene exactamente 10 tupias y conside re el efecto de "UPDATE tupia *t*" sobre *R* si el predicado de la varrel *R* indica que *R* debe contener por lo menos 10 tupias.
   - En forma similar, nunca se realizan procedimientos disparados "a la mitad de" cualquier actualización dada (de hecho, se realizan al final, inmediatamente antes de la verificación del predicado de la varrel).
   - La forma abreviada requiere de un ligero ajuste en el caso de las vistas de proyección (vea más adelante en esta sección).
6. Todas las actualizaciones sobre vistas deben ser implementadas por el mismo tipo de actua lizaciones sobre las varrels subyacentes. Esto es, los INSERTS se transforman en INSERTS y los DELETES en DELETES (gracias al punto anterior, podemos ignorar los UPDATEs). Si por el contrario, suponemos que existe cierta clase de vista (digamos una vista de unión) para la cual (digamos) los INSERTS se transforman en DELETES, ¡entonces debemos de ducir que los INSERTs *sobre una varrel base* en ocasiones también deben transformarse en DELETEs! Llegamos a esta conclusión debido a que (como ya señalamos en el punto 2 anterior) la varrel base *B* es idéntica en la semántica a la vista de unión *V* = *B* UNION *B*. También podemos aplicar un argumento similar a todas las demás clases de vistas (restric ción, proyección, intersección, etcétera). La idea de que un INSERT sobre una varrel base pudiera ser en realidad un DELETE, la tomamos como evidentemente absurda; de ahí nues tra posición de que (de nuevo) los INSERTs se transforman en INSERTs y los DELETE en DELETEs.
7. En general, las reglas de actualización, cuando se aplican a una determinada vista V, especi ficarán las operaciones que se van a aplicar a las varrels cuyos términos definen a *V*. Y dicha reglas deben funcionar correctamente, aun cuando esas mismas varrels sean también vistas. En otras palabras, las reglas deben permitir su *aplicación recursiva*. Por supuesto, si por algu na razón falla un intento por actualizar una varrel subyacente, fallará también la actualiza ción original; es decir, las actualizaciones sobre vistas son del tipo todo o nada, tal como en el caso de las actualizaciones sobre varrels base.

8. Las reglas no pueden dar por hecho que la base de datos está bien diseñada (es decir, tota mente normalizada; vea los capítulos 11 y 12). Sin embargo, en ocasiones podrían prod un resultado ligeramente sorpresivo si la base de datos *no* está bien diseñada, un hecho (en sí mismo puede verse como un argumento adicional para apoyar un buen diseño. En[1] siguiente subsección daremos un ejemplo de un "resultado ligeramente sorpresivo".
9. No debe haber una razón *a primera vista* para permitir algunas actualizaciones y otras (por ejemplo, los DELETEs pero no los INSERTs) sobre una vista determinada.
10. El INSERT y el DELETE deben ser inversos entre sí, en la medida de lo posible.

Le recordamos otra idea importante. Como explicamos en el capítulo 5, las operacionesn cionales (en particular las actualizaciones relaciónales) siembre se dan **en el nivel de** conjunt un conjunto que contiene una sola tupia es sólo un caso especial. Lo que es más, en ocasione *requieren* actualizaciones de múltiples tupias (es decir, algunas actualizaciones no pueden si larse como una serie de operaciones de una sola tupia). Y en general, esta observación es tanto para las varrels base como para las vistas. Por razones de simplicidad, en su mayoría presentaremos nuestras reglas de actualización en términos de operaciones de una sola tupi; no pierda de vista el hecho de que la consideración de operaciones de una sola tupia es solamente una simplificación y de hecho, en ciertos casos, una simplificación exagerada.

A continuación consideramos uno por uno los operadores del álgebra relacional, comen» do con unión, intersección y diferencia. *Nota:* En particular, en estos tres primeros casos dam por hecho que tratamos con una vista cuya expresión de definición es de la forma A UN! o A INTERSECT *Bo A* MINUS *B* (según corresponda), en donde *A* y *B* son a su vez expresión) relaciónales (es decir, no necesariamente denotan varrels base). Las relaciones denotadas y *B* deben ser del mismo tipo de relación. Los predicados de varrel correspondientes son f *PB,* respectivamente.

Unión

Aquí tenemos la regla de INSERT para A UNION *B:*

- **INSERT:** La nueva tupia debe satisfacer *PA* o *PB* o ambos. Si satisface *PA,* se inserta< A; observe que este INSERT podría tener el efecto lateral de insertar la tupia también en *B.* Si satisface *PB,* se inserta en *B,* a menos que ya haya sido insertada en *B* como un efect lateral de la inserción en A.

  Nota: El procedimiento específico en el que esta regla está enunciada ("insertar en A luego insertar en *B")* debe entenderse solamente como una simplificación pedagógica;nc significa que el DBMS en realidad deba realizar los INSERTs en la secuencia enunciada De hecho, el principio de simetría número 3 de la subsección inmediata anterior así lo implica, ya que ni *A* ni *B* tienen precedencia entre sí. Se aplican observaciones similares a muchas de las reglas explicadas en las secciones siguientes.

---

* Varias de las reglas y ejemplos que se explican en las secciones siguientes, se refieren a la posibilidad de efectos laterales. Ahora, es bien sabido que por lo regular los efectos laterales son indeseables; sin embargo, la idea es que esos efectos laterales podrían ser inevitables si *A y B* representan subconjuntos traslapados de la misma varrel subyacente, como será frecuentemente el caso en las vistas de unión, intersección y **difere** cia. Lo que es más, los efectos laterales en cuestión son (por esta vez) necesarios, en vez de indeseables.

Explicación: La nueva tupia debe satisfacer por lo menos uno de los predicados *PA* o *PB,* ya que en caso contrario no calificaría para su inclusión en *A* UNION *B:* es decir, no satisfaría el predicado de la varrel —o sea *(PA)* OR *(PB)*— para *A* UNION *B.* (También damos por hecho, aunque en realidad no es estrictamente necesario, que la nueva tupia no debe aparecer actualmente en *A* ni en *B,* ya que de otro modo estaríamos intentando insertar una tupia que ya existe.) Al dar por hecho que se satisfacen los requerimientos anteriores, la nueva tupia es insertada en *AotnB* según corresponda lógicamente (quizás en ambas).

Ejemplos: Sea la vista UV definida como sigue:

```
VAR UV VIEW
  ( V WHERE STATUS > 25 ) UNION ( V WHERE CIUDAD = 'París' )
```

La figura 9.2 muestra un posible valor para esta vista, de acuerdo con nuestros valores de datos usuales.

UV

V#	PROVEEDOR	STATUS	CIUDAD
V2	Jones	10	París
V3	Blake	30	París
V5	Adams	30	Atenas

Figura 9.2 Vista UV (valores de ejemplo).

- Sea la tupia a insertar (V6,Smith,50,Roma).* Esta tupia satisface el predicado de V WHERE STATUS > 25 pero no el predicado de V WHERE CIUDAD = 'París'. Por lo tanto, es in sertada en V WHERE STATUS > 25. Debido a las reglas con respecto a INSERT sobre una restricción (que son bastante obvias, como lo verá más adelante en esta sección), el efecto consiste en insertar la nueva tupia en la varrel base de proveedores y por lo tanto hacer que la tupia aparezca como se desea en la vista.
- Sea ahora la tupia a insertar (V7,Jones,50,París). Esta tupia satisface el predicado de V WHERE STATUS > 25 y el predicado de V WHERE CIUDAD = Taris'. Por lo tanto es insertada lógicamente en ambas restricciones. Sin embargo, su inserción en una restricción tiene de todos modos el mismo efecto que su inserción en la otra, así que no hay necesidad de realizar de manera explícita el segundo INSERT.

Ahora, suponga que VA y VB son dos varrels *base* distintas: VA representa a los proveedores con un status > 25 y VB representa a los proveedores en París (vea la figura 9.3); suponga que la vista UV está definida como VA UNION VB, y considere de nuevo los dos INSERTs de ejemplo que explicamos anteriormente. Insertar la tupia (V6,Smith,50,Roma) en la vista UV hará que esa tupia sea insertada en la varrel base VA (supuestamente como se requiere). Sin embargo, insertar la tupia (V7, Jones,50,París) en la vista UV hará que la tupia sea insertada en *ambas* varrels base.

---

*Por motivos de legibilidad, a lo largo de esta sección adoptamos esta notación simplificada.

VA

V#	PROVEEDOR	STATUS	CIUDAD
V3	Blake	30	París
V5	Adams	30	Atenas

VB

V#	PROVEEDOR	STATUS	CIUDAD
V2	Jones	10	París
V3	Blake	30	París

Figura 9.3 Varrels base VA y VB (valores de ejemplo).

Este resultado es lógicamente correcto, aunque podríamos decir que va contra la intuición (es un ejemplo de lo que en la sección anterior llamamos un "resultado ligeramente sorpresivo"), *Es nuestra posición que dicha sorpresa puede ocurrir solamente si la base de datos ¡ • diseñada.* En particular, también creemos que un diseño que permite que una misma tupia aparezca en dos varrels base distintas (por ejemplo, para satisfacer el predicado de ambas) espoi definición un mal diseño. Detallaremos esta postura un tanto controversial, en el capítulo 12. sección 12.6.

Pasemos ahora a la regla de DELETE para *A* UNION *B:*

- **DELETE:** Si la tupia a eliminar aparece en *A,* será eliminada de *A* (observe que este DELETE podría tener el efecto lateral de eliminar también la tupia de *B).* Si (aún) aparece en *B.* sera eliminada de *B.*

Dejamos los ejemplos para ilustrar esta regla como ejercicios. Observe que eliminar u tupia de *A o B* podría provocar un DELETE en cascada o la ejecución de algún procermiento disparado.

Por último, la regla de UPDATE:

- **UPDATE:** La tupia por actualizar debe ser tal, que la versión actualizada satisfaga*PA oPB,* o ambos. Sí la tupia a actualizar aparece en *A,* será eliminada de *A sin* ejecutar ningún pro cedimiento disparado (DELETE en cascada, etcétera) que normalmente ocasionaría **dicho** DELETE, y sin verificar tampoco el predicado de varrel de *A.* Observe que este DELETÍ podría tener el efecto lateral de eliminar también la tupia de *B.* Si la tupia (todavía) aparece en *B,* será eliminada de *B* (de nuevo, sin realizar ningún procedimiento disparado ni verifi cación de predicado de varrel). Enseguida, si la versión actualizada de la tupia satisface *PA,* será insertada en *A* (note que este INSERT podría tener el efecto lateral de insertar también la tupia en *B).* Por último, si la versión actualizada satisface *PB,* será insertada en *B,* a menos que ya haya sido insertada en *B* como un efecto lateral de su inserción en *A.*

En esencia, esta regla de UPDATE consiste en la regla de DELETE seguida por la regla de INSERT; salvo que, como señalamos, después del DELETE no se ejecutan procedimientos disparados ni verificaciones de predicado (todo procedimiento disparado asociado con el UPDATE es ejecutado conceptualmente después de hacer todas las eliminaciones y justo antes de las verificaciones de predicado).

Vale la pena señalar que una consecuencia importante de tratar a los UPDATEs de esta forma. es que un determinado UPDATE puede hacer que una tupia *emigre* de una varrel a otra. Poi ejemplo, dada la base de datos de la figura 9.3, la actualización de la tupia (V5,Adams,30,Atenas)

en la vista UV a (V5,Adams,15,París), eliminará de VA la tupia V5 anterior e insertará la nueva tupia V5 en VB.

Intersectar

Ahora bien, aquí están las reglas para actualizar A INTERSECT *B.* Esta vez, enunciaremos simplemente las reglas sin mayor explicación (éstas siguen el mismo patrón general que la regla de unión), salvo para señalar que el predicado de A INTERSECT *B* es *(PA)* AND *(PB).* Dejamos los ejemplos como un ejercicio, para ilustrar los diferentes casos.

- **INSERT:** La nueva tupia debe satisfacer tanto *PA* como *PB.* Si ésta no aparece actualmente en A, será insertada en *A* (observe que este INSERT podría tener el efecto lateral de insertar también la tupia en *B).* Si (todavía) no aparece en *B,* será insertada en *B.*
- **DELETE:** La tupia a eliminar será eliminada de A (observe que este DELETE podría tener el efecto lateral de eliminar también la tupia de *B).* Si (aún) aparece en *B,* será eliminada den.
- **UPDATE:** La tupia a actualizar debe ser tal, que la versión actualizada satisfaga tanto *PA* como *PB.* La tupia es eliminada de A sin ejecutar ningún procedimiento disparado ni veri ficación de predicado (observe que este DELETE podría tener el efecto lateral de eliminarla también de *B);* si (todavía) aparece en *B,* será eliminada de *B,* de nuevo, sin ejecutar ningún procedimiento disparado ni verificación de predicado. Después, si la versión actualizada de la tupia no aparece actualmente en A, será insertada en A (observe que este INSERT podría tener el efecto lateral de insertar también la tupia en *B).* Si (todavía) no aparece en *B,* será insertada en *B.*

Diferencia

Aquí están las reglas para actualizar A MINUS *B* (el predicado de varrel es *(PA)* AND NOT *(PB)):*

- **INSERT:** La nueva tupia debe satisfacer *PA* y no *PB.* Es insertada en A.
- **DELETE:** La tupia a eliminar es eliminada de A.
- **UPDATE:** La tupia a actualizar debe ser tal, que la versión actualizada satisfaga *PA* y no *PB.* La tupia es eliminada de A sin ejecutar ningún procedimiento disparado ni verificación de predicado de varrel; la versión actualizada será insertada entonces en A.

Restringir

Sea A WHERE *p* la expresión que define la vista Vy sea *PA* el predicado para A; entonces el predicado de *V* es *(PA)* AND *(p).* Por ejemplo, el predicado de la restricción V WHERE CIUDAD = 'Londres' es (PV) AND (CIUDAD = 'Londres'), donde PV es el predicado de proveedores. Entonces, aquí tenemos las reglas para actualizar A WHERE *p:*

- **INSERT:** La nueva tupia debe satisfacer tanto *PA* como *p.* Es insertada en A.
- **DELETE:** La tupia a eliminar es eliminada de A.

■ **UPDATE:** La tupia a actualizar debe ser tal, que la versión actualizada satisfaga tanto a *PA* como a/?. La tupia es eliminada de *A* sin ejecutar ningún procedimiento disparado ni verificación de predicado. La versión actualizada es insertada en *A.*

Ejemplos: Sea la vista LV definida como

```
VAR LV VIEW
    V WHERE CIUDAD ■ 'Londres¹ ;
```

La figura 9.4 presenta un valor de ejemplo para esta vista.

LV

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V4	Clark	20	Londres

Figura 9.4 Vista LV (valores de ejemplo).

Un intento de insertar la tupia (V6,Green,20,Londres) en LV tendrá éxito. La nueva tupia es insertada en la varrel V y por lo tanto, será insertada también en la vista LV.

Un intento de insertar la tupia (VI,Green,20,Londres) en LV fracasará, ya que viola el predicado de la varrel V (y por lo tanto, también de LV); viola específicamente la restricción de unicidad sobre la clave candidata {V#}.

Un intento de insertar la tupia (V6,Green,20,Atenas) en LV fracasará, ya que viola la restricción CIUDAD = 'Londres'.

Un intento de eliminar la tupia (VI ,Smith,20,Londres) de LV tendrá éxito. La tupia es eliminada de la varrel V y por lo tanto también será eliminada efectivamente de la vista LV.

Un intento de actualizar la tupia (Vl,Smith,20,Londres) a (V6,Green,20,Londres) en LV, tendrá éxito. Un intento de actualizar la misma tupia (VI,Smith,20,Londres) ya sea a (V4,Smith,20,Londres) o bien a (Vl,Smith,20,Atenas) fracasará (en cada caso, indique por qué).

Proyectar

Una vez más, comenzamos con una explicación sobre el predicado relevante. Estén los atributos de la varrel *A* (con predicado *PA)* divididos en dos grupos disjuntos, digamos *X* y *Y.* Considere a *X* como un solo atributo *compuesto y &Y* como otro, y considere la proyección de *A* sobre *X, A{X).* Sea *{X.x}* una tupia de esa proyección. Entonces, debe ser claro que el predicado para esa proyección es básicamente "Hay un cierto valor *y* entre los valores del dominio de *Y,* tal que la tupia *{X:x,Y:y}* satisface *PA".* Por ejemplo, considere la proyección de la varrel V sobre V#, PROVEEDOR Y CIUDAD. Toda tupia *(v,n,c)* que aparece en esa proyección es tal que existe un valor de status *t* tal que la tupia *(v,n,t,c)* satisface el predicado de la varrel V.

ri-

Entonces, aquí tenemos las reglas para actualizar *A {X}:*

- **INSERT:** Sea *(x)* la tupia a insertar. Sea *y* el valor predeterminado de *Y* (es un error si no existe dicho valor predeterminado; es decir si *Y* tiene "no se permiten valores predetermi nados")* La tupia *(x,y)* —la cual debe satisfacer *PA*— es insertada en *A.*

  Nota: Por lo reguíar (aunque noirivariablemente), los atributos de clave candidata no *tendrán un valor predeterminado* (vea eí capítulo 18). Como consecuencia, una proyección que no incluya todas las claves candidatas de la varrel subyacente, por lo regular no permitirá los INSERTs.
- **DELETE:** Todas las tupias de *A* con el mismo valor *X* que la tupia a eliminar de *A {X}* son eliminadas de *A.*

  Nota: En la práctica, es generalmente necesario que *X* incluya por lo menos una clave candidata de A; de modo que la tupia a eliminar de *A{X]* corresponda exactamente a una tupia de *A.* Sin embargo, no existe una razón lógica para hacer de éste un requerimiento estricto. También se aplican observaciones similares en el caso de UPDATE, vea lo siguiente.
- **UPDATE:** Sea *(x)* la tupia a actualizar y sea *(x')* la versión actualizada. Sea *a* una tupia de *A* con el mismo valor *x* de *X,* y sea *y* el valor de *Y* en *a.* Todas estas tupias *a* son eliminadas de *A* sin ejecutar ningún procedimiento disparado ni verificación de predicado. Entonces para cada uno de estos valores *y,* la tupia *(x',y)* —que debe satisfacer *PA*— es insertada en *A.*

  Nota: Aquí es donde aparece el "ligero ajuste" con respecto a la proyección que mencionamos en el principio número 5, en la subsección "Hacia un mecanismo de actualización de vistas". Observe específicamente que el paso "INSERT" final en la regla de UPDATE *declara de* nuevo *el* valor *y* anterior en cada tupia que es insertada; *no* lo reemplaza por el valor predeterminado aplicable, como lo haría un INSERT independiente.

Ejemplos: Sea la vista VC definida como

```
VC { V#, CIUDAD }
```

La figura 9.5 presenta un valor de ejemplo para esta vista.

ve

V#	CIUDAD
V1	Londres
V2	París
V3	París
V4	Londres
V5	Atenas

Figura 9.5 Vista VC (valores de ejemplo).

*Como implica este enunciado, aquí damos por hecho que (igual que en SQL) hay un medio disponible para especificar valores predeterminados para los atributos de las varrels base. La sintaxis adaptable de **Tutorial D** podría tomar la forma de una nueva cláusula en la definición de la varrel base, digamos DEFAULT *(<lista de especificaciones de valores predeterminados separados con comas* >), en donde cada *Kespecificación de valor predeterminado*> toma ía forma <*nombre de atributo*> <*valor predeterminado*>. Por ejemplo, en la definición de la varrel base de proveedores V, podríamos especificar DEFAULT (STATUS 0, CIUDAD ' ')•

- Un intento de insertar la tupia (V6,Atenas) en VC, tendrá éxito y tendrá el efecto de inser tar la tupia (V6,n,í, Atenas) en la varrel V; donde[1] *n* y *t* son los valores predeterminados de los atributos PROVEEDOR y STATUS, respectivamente.
- Un intento de insertar la tupia (VI,Atenas) en VC fracasará, ya que viola el predicado de varrel de V (y por lo tanto, también de VC); viola específicamente la restricción de unici dad de la clave candidata {V#}.
- Un intento de eliminar la tupia (VI,Londres) de VC tendrá éxito. La tupia de VI es elimi nada en la varrel V.
- Un intento de actualizar la tupia (VI,Londres) a (VI,Atenas) en VC, tendrá éxito; el efecto será actualizar la tupia (Vl,Smith,20,Londres) de la varrel V a (Vl,Smith,20,Atenas). Ob serve que *no la actualiza* a (Vl,«,í,Atenas); donde *n* y *t* son los valores predeterminados aplicables.
- Un intento de actualizar la misma tupia (VI,Londres) a (V2,Londres) en VC, fracasará (indique exactamente por qué).

Dejamos como ejercicio la consideración del caso en que la proyección no incluye una clave candidata de la varrel subyacente; por ejemplo, la proyección de la varrel V sobre STATUS y CIUDAD.

Extender

Sea la expresión que define la vista *V* como sigue

```
EXTEND A ADD exp AS X
```

(donde, como de costumbre, el predicado de A es PA). Entonces, el predicado *PE* de *V* es

```
PA ( a ) AND e.X = exp ( a )
```

En este caso, *e* es una tupia de *V* y *a* es la tupia que queda cuando se quita el componente *X* de *e* (es decir, en general *a* es la proyección de *e* sobre todos los atributos de *A).* En lenguaje natural (simulado):

> Toda tupia *e* en la extensión es tal, que (1) la tupia *a* que se deriva de *e* al proyectar al com- ponente *X hacia fuera,* satisface *PA* y (2) ese componente *X* tiene un valor igual al resultado de aplicar la expresión *exp* a esa tupia *a.*

Entonces, aquí tenemos las reglas de actualización:

- **INSERT:** Sea *e* la tupia a insertar; *e* debe satisfacer *PE.* La tupia *a* que se deriva de *e* al proyectar el componente *X* hacia afuera, es insertada en *A.*
- **DELETE:** Sea *e* la tupia a eliminar. La tupia *a* que se deriva de *e* al proyectar el compo nente *X* hacia afuera, es eliminada de *A.*
- **UPDATE:** Sea *e* la tupia a actualizar y sea *e'* la versión actualizada; *e'* debe satisfacer *PE.* La tupia *a* que se deriva de *e* al proyectar el componente *X* hacia afuera, es eliminada de *A*

sin ejecutar ningún procedimiento disparado ni verificación de predicado. La tupia *a'* que se deriva de *e'* al proyectar el componente *X* hacia afuera, es insertada en *A*.

Ejemplos: Esté la vista GPX definida como

```
EXTEND P ADD ( PESO * 454 ) AS PSGR
```

La figura 9.6 presenta un valor de ejemplo para esta vista.

GPX

P#	PARTE	COLOR	PESO	CIUDAD	PSGR
P1	Tuerca	Rojo	12.0	Londres	5448.0
P2	Perno	Verde	17.0	París	7718.0
P3	Tornillo	Azul	17.0	Roma	7718.0
P4	Tornillo	Rojo	14.0	Londres	6356.0
P5	Leva	Azul	12.0	París	5448.0
P6	Engrane	Rojo	19.0	Londres	8626.0

Figura 9.6 Vista GPX (valores de ejemplo).

Un intento de insertar la tupia (P7,Engrane,Rojo,12,París,5448) tendrá éxito y tendrá el efecto de insertar la tupia (P7,Engrane,Rojo,12,París) en la varrel P.

Un intento de insertar la tupia (P7,Engrane,Rojo,12,París,5449) fracasará (¿por qué?). Un intento de insertar la tupia (Pl,Engrane,Rojo,12,París,5448) fracasará (¿por qué?).

Un intento de eliminar la tupia de Pl tendrá éxito y tendrá el efecto de eliminar la tupia Pl en la varrel P.

Un intento de actualizar la tupia de Pl a (Pl,Tuerca,Rojo,10,París,4540) tendrá éxito; el efecto será actualizar la tupia (Pl,Tuerca,Rojo,12,Londres) de la varrel P a (Pl,Tuerca, Rojo,10,París).

Un intento de actualizar esa misma tupia a una de P2 (sin cambiar los demás valores) o a una en que PSGR no sea igual a 454 veces el valor PESO fracasará (indique en cada caso ¿por qué?).

Juntar

La mayoría de los tratamientos anteriores al problema de actualización de vistas —incluyendo a los de las cinco primeras ediciones de este libro y en otros libros de mi autoría— han argumentado que la posibilidad de actualización de una junta dada depende, por lo menos en parte, de si la junta es uno a uno, uno a muchos o muchos a muchos. En contraste con estos tratamientos previos, ahora afirmamos que las juntas *siempre* son actualizables. Más aún, las reglas son idénticas en los tres casos y son en esencia bastante directas. Lo que hace posible esta afirmación (tan asombrosa como podna parecer a primera vista) es la nueva perspectiva sobre el problema que sustenta la adopción de *la regla de oro,* como explicaremos enseguida.

En general, el objetivo del soporte de vistas siempre ha sido hacer que las vistas se parezcan tanto como sea posible a las varrels base; y de hecho, éste es un objetivo loable. Sin embargo:

- Por lo regular damos por hecho (de manera implícita) que siempre es posible actualizar una tupia individual de una varrel base independientemente de las demás tupias en ella.
- Al mismo tiempo, hemos podido apreciar (de manera explícita) que *no siempre* es posible *actualizar una tupia individual de una vista* independientemente de ias demás tupias de la misma.

Por ejemplo, en la referencia [11.2] Codd muestra que no es posible eliminar sólo una tupia de una cierta junta, ya que el efecto sería dejar una relación que "no sena en absoluto la junta de dos relaciones" (lo que significa que no sería posible que el resultado satisficiera el predicado de varrel de la vista). Y el enfoque para tales actualizaciones de vistas ha sido históricamente rechazarlas siempre, sobre la base de que es imposible hacer que luzcan completamente como actualizaciones de varrels base.

Nuestro enfoque es más bien diferente. Para ser específicos, reconocemos que incluso con una varrel base no siempre es posible actualizar tupias individuales independientemente del resto. Por lo tanto, en general aceptamos esas actualizaciones de vistas que históricamente han sido rechazadas, interpretándolas en una forma obvia y lógicamente correcta, para aplicarlas a las varrels subyacentes. Lo que es más, las aceptamos con un reconocimiento total del hecho de que actualizar dichas varrels subyacentes podría tener efectos laterales sobre la vista; *sin embargo, los efectos laterales son necesarios para evitar la posibilidad de que la vista pueda violar su propio predicado.*

Superado este preámbulo, pasemos a lo específico. A partir de este punto, primero definiremos nuestros términos, luego presentaremos las reglas para actualizar vistas de junta y posteriormente consideraremos las implicaciones de dichas reglas para cada uno de los tres casos (uno a uno, uno a muchos y muchos a muchos).

Considere la junta *J = A* JOIN *B,* donde (como en el capítulo 6, sección 6.4) las varrels *A, By J* tienen los encabezados *{X, Y},* {*Y,Z]* y *{X, Y,Z),* respectivamente. Sean *PA* y *PB* los predicados de *A* y *B,* respectivamente. Entonces, el predicado *PJ* de *J* es

PA (a) AND *PB* (*b*)

donde para una determinada tupia y de la junta, *a* es "la porción *A" dej* (es decir, la tupia que se deriva dey al proyectar el componente Z hacia afuera) y *b* es "la porción *B" dej* (es decir, la tupia que se deriva *dej* al proyectar el componente *X* hacia afuera). En otras palabras:

Toda tupia de la junta es tal, que la porción *A* satisface *PA* y la porción *B* satisface *PB.* Por

ejemplo, el predicado de la junta de las varrels V y VP sobre V# es como sigue:

Toda tupia *(v,n,t,c,p,q)* en la junta es tal que la tupia *(v,n,t,c)* satisface el predicado de V y la tupia *(v,p,q)* satisface el predicado de VP.

Entonces, aquí tenemos las reglas de actualización:

- **INSERT:** La nueva tuplay debe satisfacer *PJ.* Si la porción *A* de y no aparece en *A,* será insertada en *A.* * Si la porción *B* de / no aparece en *B,* será insertada en *B.*
- **DELETE:** La porción *A* de la tupia es eliminada de A y la porción *B* es eliminada de *B.*
- **UPDATE:** La tupia a actualizar debe ser tal, que la versión actualizada satisfaga *PJ.* La por ción *A* es eliminada de *A,* sin ejecutar ningún procedimiento disparado ni verificación de predicado; y_la porción *B* es eliminada de *B,* de nuevo sin ejecutar ningún procedimiento disparado ni verificación de predicado. Entonces, si la porción *A* de la versión actualizada de la tupia no aparece en *A,* será insertada en *A.* Si la porción *B* no aparece en *B,* será in sertada en *B.*

Examinemos ahora las implicaciones de estas reglas para los tres casos diferentes.

Caso 1 (uno a uno): Observe primero que aquí el término "uno a uno" sería más preciso como "(cero o uno) a (cero o uno)". En otras palabras, hay en efecto una restricción de integridad que garantiza que para cada tupia de *A* exista como máximo una tupia correspondiente en *B* y *vice-versa,* lo que implica que el conjunto de atributos y sobre los que se realiza la junta debe ser una *superclave* tanto de *A* como de *B.* (Si desea refrescar su memoria con respecto a las superclaves, consulte el capítulo 8, sección 8.8.) *Ejemplos:*

- Como un primer ejemplo, le invitamos a considerar el efecto de las reglas anteriores sobre la junta de la varrel de proveedores V para sí misma, (únicamente) sobre los números de proveedor.
- A manera de segundo ejemplo, suponga que tenemos otra varrel VR con los atributos V# y REST, en donde V# identifica a un proveedor y REST identifica el restaurante favorito de ese proveedor. Suponga que no todos los proveedores en V aparecen en VR. Considere el efecto de las reglas de actualización de junta sobre V JOIN VR. ¿Cuál sería la diferencia si algún proveedor pudiera aparecer en VR y no en V?

Caso 2 (uno a muchos): Aquí, el término "uno a muchos" sería más preciso como "(cero o uno) a (cero o más)". En otras palabras, hay en efecto una restricción de integridad que garantiza que para cada tupia de *B* haya como máximo una tupia coincidente en *A.* En general, esto significa que el conjunto de atributos *Y* sobre los cuales se realiza la junta debe incluir un conjunto, digamos *K,* tal que *K* sea una clave candidata de A y una clave externa coincidente de *B. Nota:* Si el caso es de hecho el anterior, podemos reemplazar la frase "cero o uno" por "exactamente uno".

Ejemplos: Sea la vista VVP definida como

```
V JOIN VP
```

(por supuesto, ésta es una junta de clave externa con una clave candidata coincidente). La figura 9.7 presenta valores de ejemplo.

---

*Observe que este INSERT podía tener el efecto lateral de insertar también en *B* la porción B; como en el caso de las vistas de unión, intersección y diferencia antes expuestas. También aplicamos observaciones similares a las reglas de DELETE y de UPDATE; con la finalidad de ser breves, no nos ocupamos de detallar esta posibilidad en cada caso.

VVP

V#	PROVEEDOR	STATUS	CIUDAD	P#	CANT
V1	Smith	20	Londres	P1	300
V1	Smith	20	Londres	P2	200
V1	Smith	20	Londres	P3	400
V1	Smith	20	Londres	P4	200
V1	Smith	20	Londres	P5	100
V1	Smith	20	Londres	P6	100
V2	Jones	10	Paris	P1	300
V2	Jones	10	Paris	P2	400
V3	Blake	30	Paris	P2	200
V4	Clark	20	Londres	P2	200
V4	Clark	20	Londres	P4	300
V4	Clark	20	Londres	P5	400

Figura 9.7 Vista WP (valores de ejemplo).

Un intento de insertar la tupia (V4,Clark,20,Londres,P6,100) en VVP tendrá éxito y tendrá el efecto de insertar la tupia (V4,P6,100) en la varrel VP (agregando así una tupia a la vista).

Un intento de insertar la tupia (V5,Adams,30,Atenas,P6,100) en yyp tencjr,á exiit0 y ten. drá el efecto de insertar la tupia (V5,P6,100) en la varrel VP (agregando así una tupia a la vista).

Un intento de insertar la tupia (V6,Green,20,Londres,P6,100) en VVP tendrá éxito y tendrá el efecto de insertar la tupia (V6,Green,20,Londres) en la varrel V y la tupia (V6,P6,100) en la varrel VP (agregando así una tupia a la vista).

Nota: Suponga por un momento que es posible que existan tupias en VP sin una tupia correspondiente en V. Es más, suponga que la varrel VP ya incluye algunas tupias con el número de proveedor V6, aunque ninguna con el número de proveedor V6 y el número de parte P6. Entonces, el INSERT del ejemplo que acabamos de explicar tendrá el efecto de insertar algunas tupias adicionales en la vista; es decir, la junta de la tupia (V6,Green,20,Londres) con aquellas tupias ya existentes en VP del proveedor V6.

Un intento de insertar la tupia (V4,Clark,20,Atenas,P6,100) en VVP fracasará (¿por qué?).

Un intento de insertar la tupia (Vl,Smith,20,Londres,Pl,400) en VVP fracasará (¿por qué?).

Un intento de eliminar la tupia (V3,Blake,30,París,P2,200) de VVP tendrá éxito y tendrá el efecto de eliminar la tupia (V3,Blake,30,París) de la varrel V y la tupia (V3,P2,200) de la varrel VP.

Un intento de eliminar la tupia (Vl,Smith,20,Londres,Pl,300) de VVP tendrá "éxito" —vea la siguiente nota— y tendrá el efecto de eliminar la tupia (Vl,Smith,20,Londres) de la varrel V y la tupia (Vl,Pl,300) de la varrel VP.

Nota: En realidad, el efecto global de este intento de DELETE dependerá de la regla de DELETE de clave externa de envíos a proveedores. Si la regla especifica RESTRICT,

toda la operación fracasará. Si especifica CASCADE, tendrá el efecto lateral de eliminar también todas las demás tupias de VP (y por lo tanto las de VVP) del proveedor V1.

- Un intento de actualizar la tupia (Vl,Smith,20,Londres,Pl,300) a (Vl,Smith,20,Londres,Pl,400) en VVP tendrá éxito y tendrá el efecto de actualizar la tupia (Vl,Pl,300) de VPa(Vl,Pl,400).
- Un intento de actualizar la tupia (Vl,Smith,20,Londres,Pl,300) a (Vl,Smith,20,Atenas, Pl ,400) en VVP tendrá éxito y tendrá el efecto de actualizar la tupia (V1 ,Smith,20,Londres) de V a (Vl,Smith,20,Atenas) y la tupia (Vl,Pl,300) de VP a (Vl,Pl,400).
- Un intento de actualizar la tupia (Vl,Smith,20,Londres,P 1,300) a (V6,Smith,20,Londres,Pl,300) en VVP tendrá "éxito" (vea la nota siguiente) y tendrá el efecto de actualizar la tupia (Vl,Smith,20,Londres) de V a (V6,Smith,20,Londres) y la tupia (Vl,Pl,300) de VP a(V6,Pl,300).

  Nota: En realidad, el efecto global de este intento de actualización dependerá de la regla de UPDATE de clave externa de envíos a proveedores. Dejamos los detalles como ejercicio.

Caso 3 (muchos a muchos): Aquí, el término "muchos a muchos" sería más preciso como "(cero o más) a (cero o más)". En otras palabras, no hay restricción de integridad en efecto que garantice que en su lugar estemos tratando con una situación del Caso 1 o del Caso 2.
Ejemplos: Suponga que tenemos una vista definida como

```
V JOIN P
```

(junta de V y P sobre CIUDAD, una junta muchos a muchos). La figura 9.8 presenta valores de ejemplo.

- La inserción de la tupia (V7,Bruce,15,Oslo,P8,Rueda,Blanco,25) tendrá éxito y tendrá el efecto de insertar la tupia (V7,Bruce,15,Oslo) en la varrel V y la tupia (P8,Rueda,Blanco,25,Oslo) en la varrel P; agregando así la tupia especificada para la vista.

V#	PROVEEDOR	STATUS	CIUDAD	P#	PARTE	COLOR	PESO
V1	Smith	20	Londres	P1	Tuerca	Rojo	12.0
V1	Smith	20	Londres	P4	Tornillo	Rojo	14.0
V1	Smith	20	Londres	P6	Engrane	Rojo	19.0
V2	Jones	10	Paris	P2	Perno	Verde	17.0
V2	Jones	10	París	P5	Leva	Azul	12.0
V3	Blake	30	París	P2	Perno	Verde	17.0
V3	Blake	30	París	P5	Leva	Azul	12.0
V4	Clark	20	Londres	P1	Tuerca	Rojo	12.0
V4	Clark	20	Londres	P4	Tornillo	Rojo	14.0
V4	Clark	20	Londres	P6	Engrane	Rojo	19.0

Figura 9.8 La junta de V y P sobre CIUDAD.

La inserción de la tupia (V1 ,Smith,20,Londres,P7,Rondana,Rojo,5) tendrá éxito y tendrá el efecto de insertar la tupia (P7,Rondana,Rojo,5,Londres) en la varrel P; agregando asidos tupias a la vista, la tupia (Vl,Smith,20,Londres,P7,Rondana,Rojo,5), como especificamos, y también la tupia (V4,Clark,20,Londres,P7,Rondana,Rojo,5).

La inserción de la tupia (V6,Green,20,Londres,P7,Rondana,Rojo,5) tendrá éxito y tendrá el efecto de insertar la tupia (V6,Green,20,Londres) en la varrel V y la tupia (P7,Rondana, Rojo,5,Londres) en la varrel P; agregando así *seis* tupias a la vista.

La eliminación de la tupia (Vl,Smith,20,Londres,Pl,Tuerca,Rojo,12) tendrá éxito y tendrá el efecto de eliminar la tupia (Vl,Smith,20,Londres) de la varrel V y la tupia (Pl,Tuerca, Rojo,12,Londres) de la varrel P; eliminando así *cuatro* tupias de la vista.

Dejamos como ejercicio otros ejemplos.

Otros operadores

Por último, consideraremos brevemente los operadores restantes del álgebra. Primero observamos que junta 0, semijunta, semidiferencia y dividir no son primitivos; de modo que las reglas para estos operadores pueden derivarse de las de los operadores que los definen. Por lo que respecta a los otros, tenemos:

- *Renombrar:* Trivial.
- *Producto cartesiano:* como señalamos al final de la sección 6.4 del capítulo 6, el producto cartesiano es un caso especial de la junta natural (A JOIN *B* degenera en *A* TIMES *B* cuando *Ay B no* tienen atributos en común). Como consecuencia, las reglas para A TIMES *B* son sólo un caso especial de las reglas para A JOIN *B* (como lo son también —por supuesto— las reglas para *A* INTERSECT *E),*
- *Resumir:* tampoco es primitivo, está definido en términos de extender y por ello las reglas de actualización pueden derivarse de éste. *Nota:* En la práctica, es cierto que la mayoría de las actualizaciones sobre las vistas de SUMMARIZE fracasarán. Sin embargo, esto ocurre no porque estas vistas no sean actualizables de manera *inherente,* sino más bien porque los intentos de actualización infringen alguna restricción de integridad. Por ejemplo, sea la ex presión que define la vista:

  ```
  SUMMARIZE VP PER VP { V# } ADD SUM ( CANT ) AS CANTOT
  ```

  Entonces, un intento de eliminar, digamos, la tupia del proveedor VI tendrá éxito. Sin embargo, un intento de insertar, digamos, la tupia (V5,500) fracasará debido a que viola la restricción de integridad de que el valor de CANTOT debe ser igual a la suma de todos los valores individuales de CANT aplicables. También fracasará un intento de insertar la tupia (V5,0), aunque por una razón diferente (¿por qué, exactamente?).
- *Agrupar y desagrupar:* También aquí se aplican observaciones similares a las de re sumir.
- *Cierret:* Una vez más, se aplican observaciones similares en cierta forma.

9.5 INSTANTÁNEAS (UNA DESVIACIÓN)

En esta sección nos apartamos un poco del tema para explicar brevemente las **instantáneas** [9.2]. Las instantáneas tienen algunos puntos en común con las vistas,* aunque no son lo mismo. Al igual que las vistas, las instantáneas son varrels derivadas. Sin embargo, a diferencia de las vistas, éstas son reales en vez de virtuales; es decir, están representadas no sólo por su definición en términos de otras varrels, sino también (al menos conceptualmente) por su propia copia materializada de los datos. Por ejemplo:

```
VAR P2VC SNAPSHOT
     ( ( V JOIN VP ) WHERE P# ■ P#  ( 'P2' ) ) { V#, CIUDAD }
        REFRESH EVERY DAY ¡
```

Definir una instantánea es muy similar a ejecutar una consulta, salvo que:

a. El resultado de la consulta se conserva en la base de datos bajo el nombre especificado (P2VC en el ejemplo), como una *varrel sólo de lectura* (es decir, separada de la actualización periódica; vea el punto b.);
b. De manera periódica (en el ejemplo, EVERY DAY) la instantánea se **actualiza;** es decir, su valor actual es descartado, la consulta se ejecuta otra vez y el resultado de la nueva ejecución se convierte en el nuevo valor de la instantánea.

Por lo tanto, la instantánea P2VC representa los datos relevantes como eran a lo máximo 24 horas antes (pero ¿cuál es el predicado?).

La idea de las instantáneas es que varias aplicaciones (incluso, probablemente la mayoría) pueden tolerar o incluso requerir los datos "tal como están" en un momento en particular. Un caso son las aplicaciones de informes y de contabilidad; por lo regular estas aplicaciones requieren que los datos estén congelados en un momento apropiado (por ejemplo, al final de un periodo contable) y las instantáneas permiten que dicho congelamiento ocurra sin tener que evitar que otras transacciones realicen actualizaciones sobre los datos en cuestión (es decir, sobre "los datos reales"). En forma similar, podría ser necesario congelar grandes cantidades de datos para una consulta compleja o aplicación de sólo lectura; de nuevo, sin bloquear las actualizaciones. *Nota:* Esta idea se vuelve particularmente atractiva en un entorno de base de datos distribuida o de apoyo a la toma de decisiones (vea los capítulos 20 y 21, respectivamente). Observamos que las instantáneas representan un caso especial de *redundancia controlada* (vea el capítulo 1) y que la "actualización de instantáneas" es el proceso correspondiente de *propagación de la actualización* (vea de nuevo el capítulo 1).

Entonces, una definición general de instantánea lucirá similar a la siguiente:

```
VAR <nombre de varrel> SNAPSHOT <expresión relacional>
     <lista de definición de claves candidatas> REFRESH
     EVERY <periodicidad> ¡
```

---

*De hecho, en ocasiones se les denomina **vistas materializadas** (por ejemplo, vea las referencias [9.1], [9.3], [9.6], [9.14] y [9.16]). Sin embargo, esta terminología es desafortunada y en realidad no es aprobada, ya que el hecho de que si las vistas se materialicen o no, es un aspecto de la implementación y no del modelo; de hecho, en lo que respecta al modelo, las vistas *no* se materializan y una "vista materializada" es una contradicción en dichos términos.

donde <*periodicidad*> es (por ejemplo) MONTH o WEEK o DAY o HOUR o *n* MINUTES o MONDAY o WEEKDAY ... etcétera. (En particular, una especificación de la forma REFRESH [ON] EVERY UPDATE podría ser usada para conservar la instantánea permanentemente en sincronía con las varrels de las que se deriva.) Y aquí tenemos la sintaxis del correspondiente DROP:

```
DROP VAR <nombre de varrel>
```

donde, por supuesto, el <*nombre de varrel*> se refiere específicamente a una instantánea. *Nota:* Damos por hecho que el intento de eliminar una instantánea fracasará si alguna otra varrel se refiere actualmente a ella. Como alternativa, podríamos considerar extender la definición de la instantánea para incluir una vez más algún tipo de opción "RESTRICT vs. CASCADE". No consideramos aquí esta última posibilidad.

9.6 PROPIEDADES DE SQL

En esta sección resumimos el soporte de SQL para las vistas (sólo para éstas; hasta el momento de la publicación de este libro, SQL no tenía soporte para instantáneas). Primero, la sintaxis de **CREATE VIEW** es:

```
CREATE VIEW <nombre de vista> AS <expresión de tabla>
     [ WITH [ <calificador> ] CHECK OPTION ] ;
```

donde el <*calificador*> puede ser CASCADED o LOCAL y donde el primero es el predeterminado (y de hecho la única opción sensata, como se explica en detalle en la referencia [4.19]; por esta razón omitimos aquí una mayor explicación de LOCAL). *Explicación:*

1. La <*expresión de tabla*> es la expresión que define la vista. Para una explicación detallada de las expresiones de tabla de SQL, consulte el apéndice A.
2. Si WITH CHECK OPTION es especificado, significa que los INSERTs y los UPDATES sobre la vista serán rechazados sólo si violan cualquier restricción de integridad implicada por la expresión que define la vista. Por lo tanto, observe que dichas operaciones fallarán *sólo* si WITH CHECK OPTION es especificado; es decir, *no* fallarán de manera predeter minada. Usted podrá darse cuenta —por lo que dijimos en la sección 9.4— que considera mos dicho comportamiento como lógicamente incorrecto; por lo tanto, recomendaríamos ampliamente que en la práctica *siempre* especifique WITH CHECK OPTION* (vea la refe rencia [9.8]).

   Ejemplos:

```
1. CREATE VIEW BUEN_PROVEEDOR
       AS SELECT V.V#, V.STATUS, V.CIUDAD
          FROM   V
          WHERE  V.STATUS > 15
       WITH CHECK OPTION ;
```

*Es decir, si la vista es actualizable. Como veremos más adelante, a menudo las vistas en SQL no son actualizarles y WITH CHECK OPTION no es válida si la vista no es actualizable de acuerdo con SQL.

```
2. CREATE VIEW PARTEROJA
       AS SELECT P.P#, P.PARTE, P.PESO AS PS, P.CIUDAD
          FROM   P
          WHERE  P.COLOR = 'Rojo'
       WITH CHECK OPTION ;

3. CREATE VIEW PC
       AS SELECT VP.P#, SUM ( VP.CANT ) AS CANTOT
          FROM  VP GROUP BY VP.P# ;
```

A diferencia de su contraparte en la sección 9.1 (subsección "Más ejemplos"), esta vista no incluirá filas de partes que no suministre proveedor alguno. Vea la explicación del ejemplo 7.7.8 del capítulo 7.

```
4. CREATE VIEW DOS_CIUDADES
       AS SELECT DISTINCT V.CIUDAD AS CIUDADV, P.CIUDAD AS CIUDADP
          FROM  V, VP, P WHERE V.V# = VP.V# AND   VP.P# « P.P# ;

5. CREATE VIEW PARTEROJA_PESADA
       AS SELECT PR.P#, PR.PARTE, PR.PS, PR.CIUDAD
          FROM  PARTEROJA AS PR
          WHERE  PR.PS > 12.0
       WITH CHECK OPTION ;
```

Una vista existente puede ser eliminada por medio de **DROP VIEW;** cuya sintaxis es:

```
DROP VIEW <nombre de vista> <opción> ;
```

donde (al igual que en DROP TABLE y DROP DOMAIN) <*opción*> puede ser RESTRICT o CASCADE. Si se especifica RESTRICT y se hace referencia a la vista en cualquier otra definición de vista o restricción de integridad, entonces el DROP fracasará; si se especifica CASCADE, el DROP tendrá éxito y también se eliminará cualquier definición de vista o restricción de integridad a la que haga referencia.

Recuperación de vistas

Como indicamos en la sección 9.3, en la versión actual del estándar de SQL (SQL/92) está garantizado que todas las recuperaciones contra las vistas funcionen correctamente. Por desgracia esto mismo no es cierto para ciertos productos actuales, ni tampoco para versiones anteriores del estándar. Vea al final de este capítulo, el ejercicio 9.14, parte a.

Actualización de vistas

El soporte de SQL/92 para la actualización de vistas es muy limitado. En esencia, las únicas vistas que se consideran como actualizables son aquellas que se derivan de una sola tabla base mediante alguna combinación de operaciones de restringir y proyectar. Además, aun este caso sencillo es tratado en forma incorrecta debido a la falta de entendimiento de los predicados de varrel por parte de SQL y en particular al hecho de que las tablas de SQL permiten filas duplicadas.

Aquí tenemos un enunciado más preciso de las reglas de actualización de vistas de SQL/92 (esta lista está tomada de la referencia [4.19], aunque aquí está ligeramente simplificada). En SQL, una vista es actualizable cuando se aplican todas las condiciones siguientes:

1. La expresión de tabla que define el alcance de la vista es una expresión de selección; es deci no contiene inmediatamente alguna de las palabras reservadas JOIN, UNION, INTERSEC o EXCEPT.
2. La cláusula SELECT de esa expresión de selección no contiene directamente la palabra reservada DISTINCT.
3. Todo elemento de selección en esa cláusula SELECT (después de cualquier expansión necesaria de elementos de selección del "tipo asterisco") consiste en un nombre de columna posiblemente calificado (acompañado opcionalmente por una cláusula AS) que representa una referencia individual a una columna de la tabla subyacente (vea abajo e párrafo 5).
4. La cláusula FROM de esa expresión de selección contiene exactamente una referencia de tabla.
5. Esa referencia de tabla identifica ya sea a una tabla base o a una vista actualizable. *Nota:* La tabla identificada por esa referencia de tabla es la (única) tabla subyacente de la vista ac tualizable en cuestión (vea arriba el párrafo 3).
6. Esa expresión de selección no incluye una cláusula WHERE que incluya una subconsulta, que incluya a su vez una cláusula FROM, que incluya una referencia a la misma tabla tal como es referida en la cláusula FROM que mencionamos en el párrafo 4.
7. Esa expresión de selección no incluye una cláusula GROUP BY.
8. Esa expresión de selección no incluye una cláusula HAVING.

Puntos a destacar:

1. En SQL, la posibilidad de actualizar es del tipo "todo o nada", en el sentido de que es posi ble aplicar *ya sea* las tres operaciones INSERT, UPDATE y DELETE o ninguna de ellas; por ejemplo, no es posible que sea aplicable DELETE pero no INSERT (aunque algunos productos comerciales sí soportan dicha posibilidad).
2. En SQL, la operación UPDATE *puede* o *no puede* ser aplicada a una vista determina da; no es posible que dentro de la misma vista algunas columnas sean actualizables y otras no (aunque, otra vez, algunos productos comerciales van más allá del estándar a este respecto).

9.7 RESUMEN

Una **vista** es básicamente una expresión relacional con nombre; es posible considerarla como una **varrel virtual derivada.** Las operaciones contra una vista son implementadas normalmente mediante un proceso de **sustitución;** es decir, las referencias al *nombre* de la vista son reemplazadas por la expresión que *define* la vista y este proceso de sustitución funciona precisamente debido a la característica de **cierre.** Para las operaciones de **recuperación,** el proceso de sustitución funciona el 100 por ciento de las veces (por lo menos en teoría, aunque no necesariamente en los productos actuales). Para las operaciones de **actualización,** esto también funciona el

100 por ciento de las veces* (de nuevo, sólo en teoría, aunque definitivamente no lo hace en los productos actuales); sin embargo, en el caso de algunas vistas (por ejemplo, vistas definidas en términos de **resumen),** las actualizaciones por lo regular fracasarán debido a violaciones a las restricciones de integridad. También presentamos un conjunto amplio de **principios** que debe satisfacer el esquema de actualización y mostramos en detalle el funcionamiento del esquema de actualización para vistas definidas en términos de los operadores **unir, intersectar, diferencia, restringir, proyectar, juntar y extender.** Para cada uno de estos operadores, describimos las **reglas de inferencia de predicado** (de varrel) correspondientes.

También examinamos la cuestión de las vistas y la **independencia lógica de los datos.** Existen dos aspectos para tal independencia, el **crecimiento y** la **reestructuración.** Otros beneficios de las vistas comprenden (a) su capacidad para ocultar datos y por lo tanto ofrecer una cierta medida de **seguridad** y (b) su capacidad para actuar como forma abreviada y facilitar así la vida al usuario. Explicamos dos principios importantes, ***el principio de intercambiabilidad*** (el cual implica, entre otras cosas, que debemos tener la posibilidad de actualizar las vistas) y ***el principio de la relatividad de la base de datos.***

Nos desviamos por un momento para dar una breve explicación sobre las **instantáneas.** Por último, describimos (en bosquejo) los aspectos relevantes de SQL.

EJERCICIOS

9.1 Proponga versiones equivalentes, basadas en el cálculo, de las definiciones algebraicas de vistas de la sección 9.1, subsección "Más ejemplos".

9.2 Defina una vista consistente en números de proveedor y números de parte para los proveedores y partes que no estén coubicados.

9.3 Defina una vista para los proveedores de Londres.

9.4 Defina la varrel VP de la base de datos de proveedores y partes como una vista de la varrel VPY de la base de datos de proveedores, partes y proyectos.

9.5 Defina una vista sobre la base de datos de proveedores, partes y proyectos que consista en todos los proyectos (sólo los atributos de número de proyecto y ciudad) que sean abastecidos por el provee dor VI y utilicen la parte Pl.

9.6 Dada la definición de vista

```
VAR PESOPESADO VIEW
  ( ( P RENAME PESO AS PS, COLOR AS CLR )
      WHERE PS > PESO ( 14.0 ) ) { P#, PS, CLR } ;
```

muestre la forma convertida después de aplicar el proceso de sustitución a cada una de las instrucciones siguientes:

a. `RA := PESOPESADO WHERE CLR = COLOR ( 'Verde' ) ;`

b.
```
RB := ( EXTEND PESOPESADO ADD PS + PESO ( 5.3 ) AS PSP )
                                                { P#, PSP } ;
```

*Es por esto que consideramos a las vistas como *varrels* o (en otras palabras) como *variables;* por definición, las variables siempre son actualizables.

```
C. UPDATE PESOPESAOO WHERE PS = PESO ( 18.0 ) CLR := 'Blanco' ;

d. DELETE PESOPESADO WHERE PS < PESO ( 10.0 ) ;

e. INSERT INTO PESOPESADO
   RELATION { TUPLE { P# P# ( 'P99' ), PS PESO ( 12.0
                    ), CLR COLOR ( 'Morado' ) } } ;
```

9.7 Suponga que se revisa la definición de la vista PESOPESADO del ejercicio 9.6, como sigue:

```
VAR PESOPESADO VIEW
  ( ( ( EXTEND P ADD PESO * 454 AS PS ) RENAME COLOR AS CLR )
        WHERE PS > PESO ( 14.0 ) ) { P#, PS, CLR > ;
```

(es decir, ahora el atributo PS denota el peso en gramos en lugar de libras). Ahora, repita el ejercicio 9.6.

9.8 En el capítulo 8 sugerimos que en ocasiones podría ser necesario declarar claves candidatas (o posiblemente una clave primaria para una vista). ¿Por qué sería necesaria dicha propiedad?

9.9 ¿Qué extensiones al catálogo del sistema son necesarias para el soporte de vistas, tal como des cribimos en los capítulos 3 y 5? ¿Cuáles para el soporte de instantáneas?

9.10 Suponga que una determinada varrel base *R* es reemplazada por dos restricciones *Ay B* tales que A UNION *B* siempre es igual a *R* y A INTERSECT B siempre está vacía. ¿Es posible lograr la inde pendencia lógica de los datos?

9.11

a. La intersección *A* INTERSECT *B* es equivalente a la junta A JOIN *B* (esta junta es uno a uno, aunque no *estrictamente*, ya que podrían existir tupias en A sin una contraparte en *B* y *viceversa).* ¿Son consistentes con esta equivalencia las reglas de actualización que dimos en la sección 9.4 para las vistas de intersección y juntar?

b. A INTERSECT *B* también es equivalente a A MINUS (A MINUS *B)* y a *B* MINUS *(B* MINUS A). ¿Son consistentes con esta equivalencia las reglas de actualización que dimos en la sección 9.4 para las vistas de intersección y diferencia?

9.12 Uno de los principios que establecimos en la sección 9.4 fue que INSERT y DELETE deberían ser operaciones inversas entre sí (en la medida de lo posible). ¿Acatan este principio las reglas que dimos en esa sección para actualizar las vistas de unión, intersección y diferencia?

9.13 En la sección 9.2 (en nuestra explicación de la independencia lógica de los datos), planteamos la posibilidad de reestructurar la base de datos de proveedores y partes reemplazando la varrel V por sus proyecciones VNC y VT. También señalamos que dicha reestructuración no era en absoluto un asunto trivial. ¿Cuáles son las implicaciones?

9.14 Investigue cualquier producto SQL que tenga disponible.

a. ¿Puede encontrar en ese producto algunos ejemplos de recuperación de vistas que fallen?

b. ¿Cuáles son las reglas con respecto a la actualización de vistas en ese producto? (Probablemente sean menos estrictas que las que dimos en la sección 9.6.)

9.15 Considere la base de datos de proveedores y partes, pero por razones de simplicidad ignore la varrel de partes. Aquí tenemos en bosquejo dos posibles diseños para proveedores y envíos:

```
a. V { V#, PROVEEDOR, STATUS, CIUDAD }
   VP { V#, P#, CANT }

b. VVP { V#, PROVEEDOR, STATUS, CIUDAD, P#, CANT }
   XVV { V#, PROVEEDOR, STATUS, CIUDAD }
```

El diseño a. es el de costumbre. En contraste, en el diseño b., la varrel VVP contiene una tupia para cada envío —la cual da el número de parte y cantidad aplicables y todos los detalles del proveedor— y la varrel XVV contiene los detalles de proveedores para los que no suministran parte alguna. (Observe que los dos diseños son equivalentes en información y que por lo tanto, ambos ilustran *el principio de intercambiabilidad).* Escriba definiciones de vistas del diseño b. como vistas del diseño a. y *viceversa.* Muestre además las *restricciones de base de datos* aplicables a cada diseño (si necesita refrescar su memoria con respecto a las restricciones de base de datos, consulte el capítulo 8). ¿Tiene alguno de los dos diseños ventajas obvias sobre el otro? De ser así, ¿por qué y cuáles son?

9.16 Ofrezca soluciones de SQL para los ejercicios 9.2 al 9.5.

9.17 Como último (e ¡importante!) ejercicio en esta parte del libro, visite de nuevo la definición del modelo relacional como lo dimos al final de la sección 3.2 del capítulo 3 y asegúrese de que ahora la entiende cabalmente.

REFERENCIAS Y BIBLIOGRAFÍA

9.1 Brad Adelberg, Héctor Garcia-Molina y Jennifer Widom: "The STRIP Rule System for Efficiently Maintaining Derived Data", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

STRIP es un acrónimo de STanford Real-time Information Processor. Emplea "reglas" —es decir, procedimientos disparados (vea, por ejemplo, la referencia [8.22])— para actualizar instantáneas (aquí llamadas *datos derivados)* siempre que ocurran cambios a los datos base subyacentes. El problema con estos sistemas en general es que si los datos base cambian con mucha frecuencia, la sobrecarga en la ejecución del cómputo de las reglas puede resultar excesiva. Este artículo describe las técnicas STRIP para reducir esa sobrecarga.

9.2 Michel Adiba: "Derived Relations: A Unified Mechanism for Views, Snapshots, and Distributed Data", Proc. 1981 Int. Conf. on Very Large Data Bases, Cannes, France (septiembre, 1981). Vea también la versión anterior "Database Snapshots", por Michel E. Adiba y Bruce G. Lindsay, IBM Research Report RJ2772 (marzo 7, 1980).

El primer artículo que propuso el concepto de instantánea. Expone tanto la semántica como la implementación. Con respecto a esta última, observe en particular que son posibles diversos tipos de "actualización diferencial" o *mantenimiento incremental,* no siempre es necesario que el sistema ejecute de nuevo toda la consulta original al momento de la actualización.

9.3 D. Agrawal, A. El Abbadi, A. Singh y T. Yurek: "Efficient View Maintenance at Data Warehouses", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Recuerde del capítulo 1, que un *data warehouse* es una base de datos que contiene datos para apoyar la toma de decisiones; es decir, instantáneas, para usar la terminología del presente capítulo (y las "vistas" del título de este artículo no son tales, sino instantáneas). Como señalamos en la nota a la referencia [9.2], es posible dar mantenimiento incremental a las instantáneas, lo cual es necesario por razones de rendimiento. Sin embargo, este mantenimiento puede conducir a problemas cuando las instantáneas se derivan de varias bases de datos distintas que se actualizan todas al mismo tiempo. Este artículo ofrece una solución a este problema.

9.4 H. W. Buff: "Why Codd's Rule No. 6 Must Be Reformulated", *ACM SIGMOD Record 17,* No. 4 (diciembre, 1988).

En 1985, Codd publicó un conjunto de doce reglas para usarse como "parte de una prueba para determinar si un producto que afirma ser completamente relacional en realidad lo es" [9.5]. Su regla No. 6 requería que todas las vistas que fueran actualizables teóricamente, fueran también actualizables por el sistema. En esta breve nota, Buff afirma que el problema general de actualización

de vistas no es de decisión, es decir, no existe un algoritmo general para determinar la posib dad de actualización (en el sentido de Codd) o no, de una vista arbitraria. Sin embargo, observe que la definición de actualización de vistas adoptada en el presente capítulo es en cierto mo diferente a la de Codd, en el sentido de que presta una atención explícita a los predicados varrel aplicables.

9.5 E. F. Codd: "Is Your DBMS Really Relational?" y "Does Your DBMS Run by the Rules[l] *Computerworld* (octubre 14 y 21, 1985).

9.6 Latha S. Colby *et al:* "Supporting Multiple View Maintenance Policies", Proc. 1997 ACM Sl(MOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Las "vistas" del título de este artículo no son tales, sino instantáneas. Existen tres grandes enfo ques para el mantenimiento de instantáneas:

1. *Inmediato:* Toda actualización a cualquier varrel subyacente dispara de inmediato una corres pondiente actualización a la instantánea.
2. *Diferido:* La instantánea se actualiza sólo cuando es consultada.
3. *Periódico:* La instantánea se actualiza en intervalos especificados (por ejemplo, diariamente

En general, la finalidad de las instantáneas es mejorar el rendimiento de las consultas a expen- sas del rendimiento de las actualizaciones; y las tres políticas de mantenimiento representan espectro de compromisos entre ambas. Este artículo investiga aspectos relacionados con el so- porte de diferentes políticas sobre distintas instantáneas en el mismo sistema, al mismo tiemp

9.7 Donald D. Chamberlin, James N. Gray e Irving L. Traiger: "Views, Authorization, and Locking in a Relational Data Base System", Proc. NCC *44,* Anaheim, Calif. Montvale, N.J.: AFIPS 1 (mayo, 1975).

Incluye un breve razonamiento del enfoque adoptado para la actualización de vistas en el proto tipo System R (y de ahí en SQL/DS, DB2, el estándar de SQL, etcétera). También vea la refe- rencia [9.15], la cual realiza la misma función para el prototipo University Ingres.

9.8 Hugh Darwen: "Without Check Option", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

9.9 C. J. Date y David McGoveran: "Updating Union, Intersection, and Difference Views" y "Upda ting Joins and Other Views", en C. J. Date, *Relational Database Writings 1991-1994.* Reading, Mass Addison-Wesley (1995). *Nota:* Al momento de la publicación de este libro, estaba en preparación una versión formal de estos artículos.

9.10 Umeshwar Dayal y Philip A. Bernstein: "On the Correct Translation of Update Operations on Relational Views", *ACM TODS 7,* No. 3 (septiembre, 1982).

Un primer tratamiento formal del problema de actualización de vistas (sólo para vistas de res tricción, proyección y junta). Sin embargo, no considera los predicados de varrel.

9.11 Antonio L. Furtado y Marco A. Casanova: "Updating Relational Views", en la referencia [17.1]. Existen dos grandes enfoques al problema de actualización de vistas. Uno (el único que expli camos en detalle en el presente libro) intenta ofrecer un mecanismo general que funcione, inde pendientemente de la base de datos específica que esté involucrada; está conducido solamente por las definiciones de las vistas en cuestión. El otro enfoque, menos ambicioso, requiere que el DBA especifique exactamente qué actualizaciones están permitidas para cada vista y cuál es su semántica, por medio de la escritura del código (procedimientos) para implementar dichas actua lizaciones en términos de las varrels base subyacentes. Este artículo investiga cada uno de estos dos enfoques (en 1985). Se incluye un amplio conjunto de referencias a trabajos previos.

9.12 Nathan Goodman: "View Update Is Practical", *InfoDB 5,* No. 2 (verano, 1990).

Una exposición muy informal del problema de actualización de vistas. Parafraseamos aquí un breve extracto de la introducción: "Dayal y Bernstein [9.10] demostraron que en esencia, no es posible actualizar vistas interesantes; Buff [9.4] demostró que no existe un algoritmo que pueda decidir si una vista arbitraria es actualizable. Parece haber pocos motivos de esperanza. [Sin embargo,] nada podría estar más lejos de la verdad. El hecho es que la actualización de vistas es tanto posible como práctica". Y el artículo continúa dando una variedad de técnicas de actualización *adecuadas.* No obstante, no se menciona la noción crucial de los *predicados de varrel.*

9.13 Arthur M. Keller: "Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins", Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Portland, Ore. (marzo, 1985).

Propone un conjunto de cinco criterios que deben satisfacer los algoritmos de actualización de vistas —sin efectos laterales, sólo cambios de un paso, sin cambios innecesarios, sin posibilidad de cambios más sencillos y sin parejas DELETE-INSERT en lugar de UPDATE— y presenta algoritmos que satisfacen esos criterios. Entre otras cosas, los algoritmos permiten la implementación de una clase de actualización mediante otra; por ejemplo, un DELETE sobre una vista podría traducirse en un UPDATE sobre la varrel base subyacente (por ejemplo, un proveedor podría ser eliminado de la vista de "Proveedores de Londres" al cambiar el valor de CIUDAD por París). Como otro ejemplo (que sin embargo excede el alcance del artículo de Keller), un DELETE sobre *V* (donde Vse define como la diferencia A MINUS *B)* podría ser implementado mediante un INSERT en *B* en lugar de un DELETE en *A.* Observe que, en virtud de nuestro principio número 6, nosotros rechazamos de manera explícita tales posibilidades en el cuerpo de este capítulo.

9.14 Dallan Quass y Jennifer Widom: "On-Line Warehouse View Maintenance", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Las "vistas" de este artículo no son tales, sino instantáneas. El artículo presenta un algoritmo para el mantenimiento de instantáneas que permite ejecutar transacciones de mantenimiento en forma simultánea a las consultas contra las instantáneas.

9.15 M. R. Stonebraker: "Implementation of Views and Integrity Constraints by Query Modification", Proc. ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif, (mayo, 1975).

Vea la nota a la referencia [9.7].

9.16 Yue Zhuge, Hector Garcia-Molina, Joachim Hammer y Jennifer Widom: "View Maintenance in a Warehousing Environment", Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif, (mayo, 1995).

Las "vistas" de este artículo no son tales, sino instantáneas. Cuando se informa de una actualización a ciertos datos subyacentes, el sitio del data warehouse tendría que emitir una consulta a la base de datos antes de poder realizar el mantenimiento de instantáneas requerido (y el lapso entre dicha consulta y la actualización de los datos base originales podría conducir a anomalías). Este artículo presenta un algoritmo para enfrentar dichas anomalías.

RESPUESTAS A EJERCICIOS SELECCIONADOS

9.1 Hemos numerado las siguientes soluciones como 9.1 *.n;* donde *n* es el número del ejemplo original de la sección 9.1. Hacemos nuestras suposiciones usuales con respecto a las variables de alcance.

9.1.1

```
VAR PARTEROJA VIEW
 ( PX.P#, PX.PARTE, PX.PESO AS PS,  PX.CIUDAD )
  WHERE PX.COLOR = COLOR ( 'Rojo'
```

```
9.1.2 VAR PC VIEW
        ( PX.P#,
          SUM ( VPX WHERE VPX.P# - PX.P#, CANT ) AS CANTOT ) ;

9.1.3 VAR DOS_CIUDADES VIEW
        ( VX.CIUDAD AS CIUDADV, PX.CIUDAD AS CIUDADP )
          WHERE EXISTS VPX ( VPX.V# = VX.V# AND
          VPX.P# = PX.P# ) ;

9.1.4 VAR PARTEROJA_PESADA VIEW
          PRX WHERE PRX.PS > PESO ( 12.0 ) ;
```

Aquí PRX es una variable de alcance que abarca a PARTEROJA.

```
9.2   VAR NO_COUBICADO VIEW
        ( V TIMES P ) { V#, P# } MINUS (
        V JOIN P ) { V#, P# } ;

9.3   VAR PROVEEDOR J-ONDRES VIEW
        ( V WHERE CIUDAD = 'Londres' ) { ALL BUT CIUDAD } ;
```

Nota: Aquí omitimos el atributo CIUDAD ya que sabemos que su valor debe ser Londres p¡ todos los proveedores en la vista. Sin embargo, observe que esta omisión significa que cualquie INSERT sobre la vista fracasará necesariamente (a menos que Londres resulte ser el valor predeterminado del atributo CIUDAD en la varrel de proveedores subyacente). En otras palabras, es prob¡ que una vista como ésta no pueda soportar en absoluto operaciones INSERT. (Como alternativa,podríamos considerar la posibilidad de definir que el valor predeterminado de CIUDAD, *para las tupias insertadas mediante esta vista,* sea Londres. Esta idea de **valores predeterminados específicos** de **vistas** requiere de más estudio.)

9.4 Aquí la cuestión es: ¿Cómo debe ser definido el atributo CANT en la vista VP? La respuesta sen sata parece ser que, para un determinado par V#-P#, debe ser la *suma* de todos los valores VPY.CANT tomada sobre todos los Y# de ese par V#-P#:

```
      VAR VP VIEW
          SUMMARIZE VPY PER VPY { V#, P# }
                        ADD SUM ( CANT ) AS CANT ;

9.5   VAR YC VIEW
        ( ( VPY WHERE V# = V# ( 'V1' ) ) { Y# } JOIN (
          VPY WHERE P# = P# ( 'P1' ) ) { Y# } ) JOIN
                                                Y { Y#, CIUDAD } ;
```

9.6 No mostramos las formas convertidas. Sin embargo, observamos que e. fallará, ya que la tupia presentada para inserción no satisface el predicado de la vista.

9.7 De nuevo e. fallará, aunque esta vez por una razón ligeramente diferente. Primero, el DBMS pro porcionará un valor predeterminado de PESO —digamos *p*— ya que el usuario no proporcionó un valor de PESO "real" (y por supuesto, no puede hacerlo). Segundo, es en extremo improbable que cualquiera que sea el valor PS que proporcione el usuario sea igual a *p* * 454, aun si llega a suceder (como no sucede en el INSERT mostrado) que ese valor PS en particular sea mayor que 14.0. Porto tanto, la tupia presentada para inserción de nuevo no satisface el predicado de la vista.

Nota: Podríamos argumentar que el valor de PESO en la nueva tupia debería ser asignado en forma adecuada al valor PS especificado *dividido* entre 454. Esta posibilidad requiere de más estudio.

9.8 La siguiente lista de motivos fue tomada de la referencia [5.7]:

- Si los usuarios van a interactuar con vistas en lugar de con varrels base, entonces es claro que esas vistas deben lucir tan parecidas a las varrels base como sea posible. De hecho, en forma ideal, el usuario ni siquiera debe saber que son vistas, pero debe tener la posibilidad de tratarlas como si en realidad fueran varrels base, gracias al *principio de la relatividad de la base de datos.* Y así como

el usuario de una varrel base necesita saber qué claves candidates tiene ésta (en general), también el usuario de una vista necesita saber qué claves candidatas tiene la vista (de nuevo, en general). Declarar esas claves de manera explícita es la manera obvia de dejar disponible esa información.

- El DBMS tal vez no puede deducir por sí mismo las claves candidatas (lo cual es ciertamente el caso de todos los DBMS en el mercado actual). Por lo tanto, es probable que las declaraciones explícitas sean el único medio disponible (esto es, para el DBA) de informar al DBMS —así como al usuario— de la existencia de dichas claves.
- Aun si el DBMS pudiera deducir por sí mismo las claves candidatas, las declaraciones explícitas permitirían al sistema verificar al menos que sus deducciones no fueran inconsistentes con las especificaciones explícitas del DBA.
- El DBA podría tener algún conocimiento que no tiene el DBMS y podría por lo tanto mejorar las deducciones del DBMS. La referencia [5.7] ofrece un ejemplo de esta posibilidad.

Y la referencia [11.3] ofrece otra razón, que en esencia es que tal característica debería proporcionar una manera sencilla y cómoda de declarar ciertas restricciones de integridad importantes que de otro modo sólo podrían ser establecidas en una forma muy ambigua.

9.9 Es obviamente imposible ofrecer una respuesta definitiva a esta pregunta. Ofrecemos las siguientes observaciones

- Cada vista y cada instantánea tendrán una entrada en la varrel del catálogo VARREL, con un valor de CLASEVR de "Vista" o "Instantánea" según corresponda.
- Cada vista tendrá también una entrada en una nueva varrel del catálogo, la cual podríamos también llamar VISTA. Esa entrada debe incluir la expresión relevante que define la vista.
- En forma similar, cada instantánea tendrá también una entrada en una nueva varrel del catálogo (INSTANTÁNEA). Esa entrada debe incluir la expresión relevante de definición. También debe incluir información relacionada con el intervalo de actualización de la instantánea.
- Otra varrel del catálogo mostrará qué vistas e instantáneas están definidas en términos de qué otras varrels. Observe que la estructura de esta varrel es en cierto modo similar a la de la varrel ESTRUCTURA_PARTES (vea la figura 4.6 del capítulo 4). Así como las partes pueden contener a otras partes, de igual modo las vistas e instantáneas pueden definirse en términos de otras vistas e instantáneas. Por lo tanto, observe que en este punto, son importantes las ideas que explicamos en la respuesta al ejercicio 7.7 del capítulo 7.

9.10 ¡Sí!, pero observe lo siguiente. Suponga que reemplazamos la varrel de proveedores V por (digamos) dos restricciones VA y VB, donde VA son los proveedores en Londres y VB son los proveedores que no están en Londres. Ahora podemos definir la unión de VA y VB en una vista denominada V. Si ahora intentamos (a través de esta vista) actualizar la ciudad de un proveedor de Londres por otra distinta —o la ciudad de un proveedor "que no es de Londres" por Londres— la implementación deberá transformar ese UPDATE a un DELETE sobre una de las dos restricciones y un INSERT sobre la otra. Ahora bien, las reglas que dimos en la sección 9.4 sí manejan correctamente este caso y de hecho, *definimos* (deliberadamente) a UPDATE como un DELETE seguido de un INSERT; sin embargo, había una suposición tácita de que la implementación usaría un UPDATE por razones de eficiencia. Este ejemplo muestra que en ocasiones no funciona transformar un UPDATE en un UPDATE; de hecho, determinar aquellos casos en los cuales sí funciona puede considerarse como una optimización.

9.11 a. ¡Sí! b. ¡Sí!

9.12 INSERT y DELETE siempre serán inversos entre sí en tanto que (a) la base de datos esté dise ñada de acuerdo con *el principio de diseño ortogonal* (vea la sección 12.6 del capítulo 12) *y* (b) DBMS soporte en forma apropiada los predicados de varrel. Sin embargo, si estas dos condición! no se satisfacen, existe la posibilidad de que no sean en absoluto inversos entre sí. Por ejemplo, si A y *B* son varrels base distintas, insertar la tupia *t* en *V* = A INTERSECT *B* podría hacer que *t* sólo fuera insertada en A (debido a que ya está presente en *B);* la eliminación posterior de / en Vhará ahora que *t* sea eliminada tanto de *A* como de *B.* (Por otra parte, eliminar *t* y después reinsertarlo preservará siem pre *el estado actual de las cosas.)* Sin embargo, observe con cuidado que esta asimetría sólo puede surgir si *t* satisface el predicado de A y aún no está, antes que nada, presente en A.

9.13 Ofrecemos los siguientes comentarios. Primero, el propio proceso de reemplazo comprende varios pasos, los cuales podríamos resumir como sigue (perfeccionaremos esta secuencia de operación en un momento).

```
/* define las nuevas varrels base */

VAR VNC BASE RELATION
      { V# V#, PROVEEDOR NOMBRE, CIUDAD CHAR }
        PRIMARY KEY { V# } ;

VAR VT BASE RELATION
      { V# V#, STATUS INTEGER }
        PRIMARY KEY { V# }

/* copia los datos a las nuevas varrels base */

INSERT INTO VNC V { V#, PROVEEDOR, CIUDAD } ;

INSERT INTO VT V { V#, STATUS } ; /* elimina la

varrel anterior •/ DROP VAR V ;
```

Ahora podemos crear la vista deseada:

```
VAR V VIEW
      VNC JOIN VT  ;
```

Ahora observamos que cada uno de los dos atributos V# (en VNC y VT) constituyen una clave esterna que hace referencia a la otra. De hecho, existe un vínculo estricto de uno a uno entre las **vant** VNC y VT; y por ello entramos en algunas dificultades "uno a uno", que explico en otra parte [13.7].

Observe también que debemos hacer algo con la clave externa de la varrel VP que hace referencia a la anterior varrel V. Sería agradable que esa clave externa pudiera en su lugar ser tomada cornord rencia a la vista V; si esto no es posible (y es típico que no lo sea en los productos actuales), entones sería mejor agregar una tercera proyección de la varrel V a la base de datos, como sigue:

```
VAR VV BASE RELATION
     { V# V# } PRIMARY KEY { V# } ;

INSERT INTO VV V { V# } ;
```

(De hecho, este diseño se recomienda en la referencia [8.10] aunque por otros motivos.) Ahora cambiamos la definición de la vista V de la siguiente manera:

```
VAR V VIEW V
      VV JOIN VNC JOIN VT ;
```

También agregamos la siguiente especificación de clave externa a las definiciones de las vanéis VNC y VT:

```
FOREIGN KEY { V# } REFERENCES VV
                  ON DELETE CASCADE
                  ON UPDATE CASCADE
```

Por último, debemos cambiar la especificación de la clave externa {V#) en la varrel VP para hacer referencia a VV en vez de a V.

Nota: La idea de permitir que una clave externa haga referencia a una vista en lugar de a una varrel base, requiere de un mayor estudio.

9.14 Con respecto a la parte a. de este ejercicio, aquí tenemos un ejemplo de una recuperación de vista que (al momento de la publicación de este libro) ciertamente falla en algunos productos SQL. Considere la siguiente definición de vista de SQL (ejemplo 3 de la sección 9.6):

```
CREATE VIEW PC AS
       SELECT VP.P#, SUM ( VP.CANT ) AS CANTOT
       FROM  VP
       GROUP BY VP.P# ;
```

Considere también el siguiente intento de consulta:

```
SELECT AVG ( PC.CANTOT ) AS PT
FROM   PC ;
```

Si seguimos el sencillo proceso de sustitución que explicamos en el cuerpo del capítulo (es decir, si intentamos reemplazar las referencias al nombre de la vista por la expresión que la define), obtenemos algo como lo siguiente:

```
SELECT AVG ( SUM ( VP.CANT ) ) AS PT
FROM  VP
GROUP BY VP.Pi» ;
```

Y ésta no es una instrucción SELECT válida, debido a que (como señalamos en la explicación que sigue al ejemplo 7.7.7 del capítulo 7) SQL no permite anidar de esta manera operadores de totales.

Aquí tenemos otro ejemplo de una consulta contra la misma vista PC que también falla en ciertos productos (en gran parte, por la misma razón):

```
SELECT PC.P#
FROM   PC
WHERE   PC. CANTOT > 500  ;
```

Precisamente debido a los problemas que ilustran estos ejemplos, ciertos productos (el DB2 de IBM es uno de ellos) en ocasiones materializan físicamente la vista (en lugar de aplicar el procedimiento más usual de sustitución) y luego ejecutan la consulta contra la versión materializada. Por supuesto, esta técnica siempre funcionará, pero está sujeta a incurrir en una penalización en el rendimiento. Es más, en el caso particular de DB2, aún se da el caso que algunas recuperaciones sobre ciertas vistas no funcionan; es decir, DB2 *no siempre* usa la materialización cuando la sustitución no funciona, ni tampoco es fácil de precisar qué casos funcionan y cuáles no. Por ejemplo, el segundo de los dos ejemplos de arriba sigue fallando en DB2 (al momento de la publicación de este libro). Para una mayor explicación, vea la referencia [4.20]. 9.15 Primero, aquí tenemos una definición del diseño b. en términos del diseño a.:

```
VAR VVP VIEW
    V JOIN VP J

VAR XVV VIEW
    V MINUS ( V JOIN VP ) { V#, PROVEEDOR, STATUS, CIUDAD } ;
```

y aquí está una definición del diseño a. en términos del diseño b.:

```
VAR V VIEW
    XVV UNION VVP { V#, PROVEEDOR, STATUS, CIUDAD } ;

VAR VP VIEW
    VVP { V#, P#, CANT } ;
```

Las restricciones de base de datos aplicables a los dos diseños pueden ser declaradas como si

```
CONSTRAINT DISEÑO A
    IS_EMPTY ( VP { V# } MINUS V { V# } ) ;

CONSTRAINT DISEÑO_B
    IS_EMPTY ( VVP { V# } INTERSECT XVV { V# } ) ;
```

(Observe que aquí la restricción DISEÑO_A ejemplifica otra forma de formular una restrici referencial.)

El diseño a. es claramente superior, por las razones que explicamos en detalle en el capítuk

9.16 Hemos numerado las siguientes soluciones como 9.16.w, en donde *n* es el número del ejercic original.

9.16.2
```
CREATE VIEW NO_COUBICADO
    AS SELECT V.V#, P.P# FROM V, P WHERE
          V.CIUDAD <> P.CIUDAD ;
```

9.16.3
```
CREATE VIEW PROVEEDORJ-ONDRES
    AS SELECT V.V#, V.PROVEEDOR, V.STATUS
          FROM   V WHERE  V.CIUDAD =
          'Londres' ;
```

9.16.4
```
CREATE VIEW VP
    AS SELECT VPY.Vl», VPY.Pf», SUM ( VPY.CANT ) AS CANT
          FROM   VPY GROUP  BY VPY.V#, VPY.P# ;
```

9.16.5
```
CREATE VIEW YC
    AS SELECT Y.Y#, Y.CIUDAD FROM   Y
          WHERE  Y.Y# IN ( SELECT VPY.Y#
                           FROM  VPY
                           WHERE VPY.V# = 'V1' )
          AND    Y.Y# IN ( SELECT VPY.Y#
                           FROM  VPY
                           WHERE  VPY.P# = 'P1 ' ) ;
```

PARTE III

DISEÑO DE BASES DE DATOS

Esta parte del libro se ocupa del tema general del diseño de bases de datos (de forma más específica, del diseño de bases de datos *relaciónales).* Podemos enunciar el problema del diseño de bases de datos de manera muy sencilla: dado cierto cuerpo de datos que deben ser representados en una base de datos, ¿cómo decidimos una estructura lógica conveniente para esos datos?; en otras palabras, ¿cómo decidimos qué varrels deben existir y qué atributos deben tener? La importancia práctica de este problema es obvia.

Antes de entrar en detalles, son necesarias algunas observaciones preliminares:

- Primero, observe que aquí nos concentramos solamente en el diseño *lógico* (o *conceptual),* no en el diseño físico. Desde luego, no estamos sugiriendo que el diseño físico no sea importante; al contrario, el diseño físico es muy importante. Sin embargo:
  a. Podemos abordar el diseño físico como una actividad de seguimiento independiente. En otras palabras, la manera "correcta" de hacer el diseño de una base de datos consiste en ha cer primero un diseño lógico (es decir, relational) minucioso y después, como un paso independiente posterior, transformar ese diseño lógico en cualquier estructura física que soporte el DBMS de destino. (Dicho de otro modo, como vimos en el capítulo 2, el dise ño físico debe derivarse del diseño lógico y no al revés.)
  b. Por definición, el diseño físico tiende a ser en cierta forma específico al DBMS (y por lo tanto no es apropiado como tema para un libro de texto general como éste). En con traste, el diseño lógico es, o debe ser, bastante independiente del DBMS y existen al gunos principio teóricos sólidos que pueden ser aplicados al problema. Y por supuesto, dichos principios definitivamente tienen un lugar en un libro de esta naturaleza.

Por desgracia, vivimos en un mundo imperfecto, y en la práctica podría darse el caso de que las decisiones de diseño hechas en el nivel físico tengan un efecto contraproducente en el nivel lógico (precisamente debido al punto, que ya hemos señalado muchas veces en este libro, de que los productos DBMS actuales soportan solamente transformaciones más bien simples entre los niveles lógico y físico). En otras palabras, podría ser necesario hacer varias iteraciones sobre el ciclo de diseño "lógico luego físico" y podría ser necesario establecer compromisos. No obstante, nos mantenemos en nuestro punto de vista original de que la forma correcta de diseñar es mediante la obtención en primer lugar del diseño lógico correcto, sin prestar ninguna atención en esta etapa a las consideraciones físicas (es decir, de rendimiento). Por lo tanto, esta parte del libro se ocupa principalmente de lo que implica "la obtención en primer lugar del diseño lógico correcto".

Aunque (como ya mencionamos) nos ocupamos principalmente del diseño *relational* es nuestra firme creencia que las ideas que explicaremos también son importantes para el di seño de bases de datos no relaciónales. En otras palabras, creemos que la manera coro de diseñar una base de datos en un sistema no relacional consiste en hacer primero un diseño lógico minucioso y después, como un paso posterior independiente, transformar ese < seño en cualquier estructura no relacional (por ejemplo, jerarquías) que soporte el DMBS de destino.

Dicho todo lo anterior, podemos también decir ahora que el diseño de base de datos siguí siendo en gran medida un arte, no una ciencia. *Existen* (para repetir) algunos principios científicos que pueden ser aplicados al problema (y dichos principios son el tema de los siguientes tres capítulos); sin embargo, existen muchos aspectos del diseño que esos principie simplemente no abordan en lo absoluto. Como consecuencia, diversos teóricos y profes nales de las bases de datos han propuesto metodologías de diseño* —algunas de ellas ba tante rigurosas, otras no tanto, pero todas ellas adecuadas hasta cierto punto— que pueden ser utilizadas como un ataque en contra de lo que hasta el momento de la publicación este libro era todavía un problema intratable; es decir, el problema de encontrar el diseño lógico "único" que sea sin lugar a dudas el adecuado. Puesto que estas metodologías sonei su mayoría adecuadas hasta cierto punto, existen pocos criterios objetivos para preferir algún enfoque sobre todos los demás. Sin embargo, en el capítulo 13 presentamos un e: foque bien conocido que sí tiene por lo menos el mérito de ser utilizado ampliamente en la práctica. En ese capítulo, también consideramos brevemente otros enfoques soportados comercialmente.

Debemos declarar explícitamente un par de suposiciones que son la base de la mayoría de las explicaciones de esta parte del libro:

a. El diseño de bases de datos no es sólo una cuestión de obtener las estructuras de dato: correctas, también la integridad de los datos es un ingrediente clave (quizás el ingre diente clave). Repetiremos y ampliaremos esta observación en muchas de las ideas que aparecen en los capítulos siguientes.
b. Nos ocuparemos en gran medida de lo que podríamos denominar diseño *independíeme de la aplicación*. En otras palabras, nos ocupamos principalmente de lo que .son lo: datos, en lugar de cómo serán *usados.* En este sentido, la independencia de la aplicación es necesaria por la muy buena razón de que normalmente —o tal vez siempre— se da el caso de que al momento del diseño no son bien conocidos todos los usos que se darán a los datos; por lo tanto, queremos un diseño que sea *robusto,* en el sentido de que no s invalidado por el advenimiento de requerimientos de la aplicación que no se previeron al momento del diseño. Para ponerlo de otra forma (y para utilizar la terminología de capítulo 2), lo que tratamos de hacer principalmente, es *obtener el esquema concepto correcto;* es decir, nos interesa producir un diseño lógico abstracto que sea indepen diente del hardware, del sistema operativo, del DBMS, del lenguaje, del usuario, etcétera. En particular, como ya explicamos, *no* estamos interesados en hacer compromisos por razones de rendimiento.

---

*E1 término *metodología* significaba originalmente el *estudio de los métodos,* pero ha venido a ser usado como "un sistema de métodos y reglas aplicables a la investigación o el trabajo en una determinada ciencia o arte" *{Chambers Twentieth Century Dictionary).*

- Anteriormente dijimos que el problema del diseño de bases de datos consistía en decidir qué varrels deben existir y qué atributos deben tener éstas. Por supuesto, esto comprende el problema de decidir qué *dominios* o *tipos* deben ser definidos también. Sin embargo, no tendremos mucho qué decir sobre este tema, ya que al momento de la publicación de este libro, parece haber pocos trabajos importantes (excepciones a lo anterior son las referencias [13.11] y [13.39]).

La estructura de esta parte es la siguiente. El capítulo 10 establece algunos fundamentos teóricos. Los capítulos 11 y 12 se ocupan de las ideas de una *mayor normalización,* las cuales se desarrollan a partir de ese trabajo previo para dar un significado formal a la afirmación de que, en cierta forma, algunos diseños son "mejores" que otros. Después, el capítulo 13 explica el *modelado semántico;* en particular, describe los conceptos del modelado "entidad/vínculo" y muestra cómo esos conceptos pueden abordar el problema del diseño de arriba hacia abajo o *top-down* (comenzando con entidades reales y terminando con un diseño relacional formal).

CAPITULO **10**

Dependencias funcionales

10.1 INTRODUCCIÓN

En este capítulo examinamos un concepto que se ha caracterizado por "no ser muy fundamental, aunque está muy cerca de serlo" [10.7]; es decir, el concepto de la **dependencia funcional.** Este concepto resulta ser de crucial importancia para diversos aspectos que explicaremos en los capítulos posteriores, incluyendo en particular la teoría de diseño de base de datos descrita en el capítulo 11.

En esencia, una dependencia funcional (abreviada DF) es *un vínculo muchos a uno* que va de un conjunto de atributos a otro dentro de una determinada varrel. Por ejemplo, en el caso de la varrel de envíos VP, existe una dependencia funcional del conjunto de atributos {V#,P#} al conjunto de atributos {CANT}. Lo que esto significa es que dentro de cualquier relación que resulte ser un valor válido de la varrel VP:

a. Para cualquier valor dado del par de atributos V# y P#, sólo existe un valor correspondiente del atributo CANT, pero
b. Muchos valores distintos del par de atributos V# y P# pueden tener (en general) el mismo valor correspondiente del atributo CANT.

Observe que nuestro valor VP usual de ejemplo (vea la figura 3.8) sí satisface estas dos propiedades.

En la sección 10.2 definimos con más precisión el concepto de dependencia funcional, distinguiendo claramente aquellas DFs que satisface una varrel dada en un momento determinado y aquellas que deben ser satisfechas en *todo* momento por esa varrel. Como ya mencionamos, resulta que las DFs proporcionan una base para un ataque científico contra diversos problemas prácticos. Y la razón por la que lo hacen es debido a que poseen un rico conjunto de propiedades formales interesantes, las cuales hacen posible tratar los problemas en cuestión de una manera formal y rigurosa. Las secciones 10.3 a 10.6 analizan en detalle algunas de esas propiedades formales y explican algunas de sus consecuencias básicas. Después, la sección 10.7 presenta un breve resumen.

Nota: Quizás desee saltarse partes de este capítulo en una primera lectura. De hecho, gran parte de lo que necesita para entender el material de los tres capítulos siguientes lo cubrimos en las secciones 10.2 y 10.3. Por lo tanto, tal vez prefiera dar por ahora una "lectura ligera" a las

r

secciones restantes y regresar a ellas más adelante cuando haya asimilado el material de los tres siguientes capítulos.

10.2 DEFINICIONES BÁSICAS

Con el fin de ilustrar las ideas de la presente sección, hacemos uso de una versión ligeramente modificada de la varrel de envíos que incluye, además de los atributos usuales V#, P# y CANT, *un atributo CIUDAD que representa la ciudad del proveedor relevante. Nos referiremos a esta* varrel modificada como VCP para evitar confusiones. La figura 10.1 muestra un posible valor de la varrel VCP.

VCP

V#	CIUDAD	P#	CANT
V1	Londres	P1	100
V1	Londres	P2	100
V2	París	P1	200
V2	París	P2	200
V3	París	P2	300
V4	Londres	P2	400
V4	Londres	P4	400
V4	Londres	P5	400

Figura 10.1 Valor de muestra de la varrel VCP.

Ahora bien, es muy importante en esta área —¡así como en muchas otras!— distinguir con claridad entre (a) el valor de una varrel dada en un determinado momento y (b) el *conjunto de todos los valores posibles* que podría tomar esa varrel en diferentes momentos. En lo que sigue, primero definiremos el concepto de dependencia funcional como se aplica al caso (a) y después lo ampliaremos para aplicarlo al caso (b). Entonces, aquí está la definición del caso (a):

■ Sea *r* una relación y sean *X* y *Y* subconjuntos arbitrarios del conjunto de atributos de *r*. Entonces decimos que *Y* es **dependiente funcionalmente** de *X* —en símbolos,

(lea "X **determina funcionalmente a** *Y'*, o simplemente "X flecha F')— si y sólo si cada valor de *X* en *r* está asociado precisamente con un valor de *Y* en *r*. En otras palabras, siempre que dos tupias de *r* coincidan en su valor *X*, también coincidirán en su valor *Y*.

Por ejemplo, la relación que muestra la figura 10.1 satisface la DF

```
{ V# } -» { CIUDAD }
```

ya que toda tupia de esa relación con un determinado valor V# tiene también el mismo valo CIUDAD. De hecho, también satisface varias otras DFs, entre ellas las siguientes:

```
{ V#, P# } -> { CANT }
{ V#, P# } -> { CIUDAD }
{ V#, P# } -» { CIUDAD, CANT }
{ V#, P# } -» { V# }
{ V#, P# } -> { V#, P#, CIUDAD, CANT }
{ V# }     -> { CANT }
{ CANT }   -> { V# }
```

(Ejercicio: Verifique lo anterior).

En ocasiones, las partes izquierda y derecha de una DF se denominan **determinante** y **dependiente,** respectivamente. Como indica la definición, el determinante y el dependiente son ambos *conjuntos* de atributos. Sin embargo, cuando uno de estos conjuntos contiene sólo un atributo, es decir, cuando es un **conjunto individual,** a menudo eliminamos las llaves y escribimos solamente,

```
V# -> CIUDAD
```

Como ya expliqué, las definiciones anteriores se aplican al "caso (a)"; es decir, a *valores* de relación individuales. Sin embargo, cuando consideramos las *variables* de relación (las vanéis i. y en particular cuando consideramos varrels *base,* por lo regular estamos interesados no **tanto** en las DFs que resultan válidas para el valor particular que tiene la varrel en un momento terminado, sino más bien en aquellas DFs que son válidas para *todos los valores posibles* de esa varrel. Por ejemplo, en el caso de VCP, la DF

```
V# -» CIUDAD
```

es válida para todos los valores posibles de VCP, ya que en cualquier momento dado un determinado proveedor tiene precisamente una ciudad correspondiente, de modo que dos tupias cualesquiera que aparezcan en VCP al mismo tiempo con el mismo número de proveedor, tambié deben tener necesariamente la misma ciudad. Además, el hecho de que esta DF sea válida "todo el tiempo" (es decir, para todos los valores posibles de VCP) es una *restricción de integridad* sobre la varrel VCP y limita los valores que esa varrel VCP puede tomar legítimamente. Aquí tenemos una formulación de dicha restricción, empleando la sintaxis del capítulo 8:

```
CONSTRAINT DF_V#_CIUDAD
   COUNT ( VCP { V# } ) = COUNT ( VCP { V#, CIUDAD } ) ;
```

La sintaxis V# —> CIUDAD puede ser considerada como una forma abreviada de esta formulación más larga.

Entonces, aquí tenemos la definición de dependencia funcional del "caso (b)"; las extensiones sobre la definición del caso (a) las mostramos en **negritas:**

- Sea *R* una **variable** de relación y sean *X* y *Y* subconjuntos cualesquiera del conjunto de atributos de *R.* Entonces, decimos que *Y* es dependiente funcionalmente de *X* en símbolos,

  (lea *"X* determina funcionalmente a I", o simplemente *"X* flecha *Y")* si y sólo si en todo **valor válido posible de *R,*** cada valor *X* está asociado precisamente con un valor *Y.* En otra palabras, **en todo valor válido posible de *R,*** siempre que dos tupias coincidan en su valor *X,* también coincidirán en su valor *Y.*

De aquí en adelante usaremos el término "dependencia funcional" en este último sentido más demandante e *independiente del tiempo,* salvo enunciados explícitos que indiquen lo contrario. Entonces, aquí tenemos algunas DFs (independientes del tiempo) que se aplican a la varrel VCP:

```
{ V#, P# } -> CANT
{ V#, P# } -> CIUDAD
{ V#, P# } -> { CIUDAD, CANT }
  V#, P#   -> V#
{ V#, P# } -> { V#, P#, CIUDAD, CANT }
{ V#, }    -> CIUDAD
```

En particular, observe que las siguientes DFs que son válidas para el valor de relación que muestra la figura 10.1, *no* son válidas "todo el tiempo" para la varrel VCP:

```
V#    -> CANT
CANT -» V#
```

En otras palabras, el enunciado que dice (por ejemplo) "todo envío de un proveedor dado tiene la misma cantidad" resulta ser cierto para el valor de relación VCP particular que muestra la figura 10.1; aunque no es cierto para todos los valores válidos posibles de la varrel VCP.

Ahora, observamos que si *X* es una clave candidata de la varrel *R,* entonces todos los atributos Fde la varrel *R* deben ser dependientes funcionalmente de *X.* (Este hecho lo mencionamos de paso en la sección 8.8 y se desprende de la definición de clave candidata.) Por ejemplo, para la varrel de partes P, tenemos necesariamente:

```
P# -> { P#, PARTE, COLOR, PESO, CIUDAD }
```

De hecho, si la varrel *R* satisface la DF *A* -» *B* y *A no* es una clave candidata,* entonces *R* involucrará cierta **redundancia.** Por ejemplo, en el caso de la varrel VCP, la DF V# —»CIUDAD implica que el hecho de que un determinado proveedor esté ubicado en una ciudad, en general aparecerá muchas veces (para una ilustración, vea la figura 10.1). Tomaremos esta idea y la explicaremos en detalle en el siguiente capítulo.

Ahora bien, aun cuando limitamos nuestra atención a las DFs que son válidas todo el tiempo, el conjunto total de DFs para una determinada varrel puede seguir siendo muy grande, tal como sugiere el ejemplo VCP. *{Ejercicio:* Enuncie todo el conjunto de DFs que satisface la varrel VCP). Lo que quisiéramos encontrar es una forma de reducir ese conjunto a un tamaño manejable; y de hecho, la mayor parte de este capítulo se ocupa exactamente de este aspecto.

¿Por qué es necesario este objetivo? Una razón, como ya mencionamos, es que las DFs representan restricciones de integridad y el DBMS necesita por ello hacer que se cumplan. Por lo tanto, dado un conjunto particular *S* de DFs, es necesario encontrar algún otro conjunto *T* que sea (de manera idónea) mucho menor que *S* y que tenga la propiedad de que cada DF en *S* esté implícita por las DFs en *T.* Si es posible encontrar dicho conjunto *T,* es suficiente con que el DBMS haga cumplir sólo las DFs en *T* y entonces, las DFs en *S* se cumplirán automáticamente. Por lo tanto, el problema de encontrar dicho conjunto *T* resulta de un considerable interés práctico.

---

*Además de que la DF no es *trivial* (vea la sección 10.3), *A* no es una *superclave* (vea la sección 10.5) y *R* contiene por lo menos dos tupias.

10.3 DEPENDENCIAS TRIVIALES Y NO TRIVIALES

Nota: En lo que resta del capítulo, abreviaremos ocasionalmente "dependencia funcional" a sólo "dependencia." De manera similar abreviaremos los términos "dependiente funcionalmente de", "determina funcionalmente", etcétera.

Una forma obvia de reducir el tamaño del conjunto de DFs con las que necesitamos tratar, es eliminando las dependencias *triviales.* Una dependencia es "trivial" cuando no existe la posibilidad de que no sea satisfecha. Sólo una de las DFs de la varrel VCP de la sección anteriores trivial en este sentido; para ser más específicos, la DF

```
{ v#, P# } -» V#
```

De hecho, una DF es **trivial** si y solamente si la parte derecha es un subconjunto (no necesariamente un subconjunto propio) de la parte izquierda.

Como su nombre implica, en la práctica las dependencias triviales no son muy interesantes; por lo regular, estamos más interesados en las dependencias **no triviales** (las cuales son —por supuesto— precisamente aquellas que no son triviales), ya que son las únicas que corresponden a restricciones de integridad "genuinas". Sin embargo, cuando tratamos con la teoría formal, debemos tomar en cuenta *todas* las dependencias, tanto triviales como no triviales.

10.4 CIERRE DE UN CONJUNTO DE DEPENDENCIAS

Ya sugerimos que algunas DFs podrían implicar a otras. Como un ejemplo sencillo, la DF

```
{ v#, P# } -> { CIUDAD, CANT }
```

implica a las dos siguientes:

```
{ V#, P# } -» CIUDAD {
V#, P# } -» CANT
```

Como un ejemplo más complejo, suponga que tenemos una varrel *R* con tres atributos A, *B* y *C,* tales que las DFs *A* —> *B* y *B* —> C son válidas para *R.* Entonces, es fácil ver que la DF *A* -> *C* también es válida para *R.* Aquí, la DF *A* —¥ *C* es un ejemplo de DF **transitiva;** decimos que *C* depende de *A transitivamente,* a través de *B.*

Al conjunto de todas las DFs implicadas por un conjunto dado *S* de DFs se le llama **cierre** de *S* y se escribe S^+. Resulta claro que necesitamos un algoritmo que (por lo menos en principio) nos permita calcular S^+ a partir de *S.* El primer ataque a este problema apareció en un artículo de Armstrong [10.1], el cual proporcionó un conjunto de **reglas de inferencia** (mejor conocidas como **axiomas de Armstrong)** mediante las cuales es posible inferir nuevas DFs a partir de las ya dadas. Dichas reglas pueden ser enunciadas en diversas formas equivalentes, de las cuales una de las más sencillas es la siguiente. Sean *A, B* y *C* subconjuntos cualesquiera del conjunto de atributos de la varrel dada *R* y acordemos escribir (por ejemplo) *AB* para representar la unión de *A* y *B.* Entonces:

1. **Reflexividad:** Si *B* es un subconjunto de *A,* entonces *A* —> *B.*
2. **Aumento:** Si *A* —> *B,* entonces *AC* -* *BC.*
3. **Transitividad:** Si *A* -» *B y B* -> C, entonces ,4 -> C.

Cada una de estas reglas puede ser demostrada directamente a partir de la definición de dependencia funcional (por supuesto, la primera es simplemente la definición de una dependencia **trivial).** Además, las reglas son **completas,** en el sentido de que dado un conjunto *S* de DFs, todas las DFs implicadas por *S* pueden derivarse de *S* utilizando las reglas. También son firmes, en el sentido de que ninguna DF adicional (es decir DFs no implicadas por *S)* puede ser derivada de esa forma. En otras palabras, las reglas pueden ser usadas para derivar precisamente el cierre S^+.

Otras reglas adicionales pueden derivarse de las tres anteriores, entre ellas las siguientes. Estas reglas adicionales pueden ser usadas para simplificar la tarea práctica de calcular S^+ a partir de *S.* (A partir de este punto, *D* es otro subconjunto arbitrario de atributos de *R.)*

4. **Autodeterminación:** *A* —> *A.*
5. **Descomposición:** Si *A* —> *BC,* entonces *A* —> *B* y *A* —> *C.*
6. **Unión:** Si *A* —> *B* y *A* -> C, entonces *A* —> *BC.*
7. **Composición:** Sí *A* -> *B y C* -» *D,* entonces *AC* -> *BD.*

Y en la referencia [10.6], Darwen demuestra la siguiente regla, a la cual llama *teorema de unificación general:*

8. Si *A* —> *B* y *C* —» *D,* entonces Au(C-B)-) *BD* (en donde "u" es unión y "-" es dife rencia de conjuntos).

El nombre "teorema de unificación general" se refiere al hecho de que varias de las reglas anteriores pueden ser consideradas como casos especiales [10.6].

Ejemplo: Suponga que tenemos la varrel *R* con los atributos *A, B, C, D, E, F,* y las DFs

```
A  —> BC
B  —¥ E
CD —» EF
```

Observe que estamos extendiendo ligeramente nuestra notación, aunque no en forma incompatible, al escribir —por ejemplo— *BC* para el conjunto de atributos *B y C* (anteriormente, *BC* habría significado la *unión* de *B y* C; donde *B y C* serían *conjuntos* de atributos). *Nota:* Si prefiere un ejemplo más concreto, tome *A* como número de empleado, *B* como número de departamento, C como número de empleado del gerente, *D* como el número del proyecto dirigido por ese gerente (único dentro del gerente), *E* como nombre del departamento y *F* como porcentaje de tiempo asignado por el gerente especificado al proyecto especificado.

Ahora mostramos que la DF *AD* —> *F* es válida para *R y* por lo tanto es un miembro del cierre del conjunto dado:

L *A* - **BC** (dado)
2. *A* - **C** (1, descomposición)
3. *AD* - **CD** (2, aumento)
4. *CD* - *EF* (dado)
5. /ID - *EF* (3 y 4, transitividad)
6. *AD* - *F* (5, descomposición)

10.5 CIERRE DE UN CONJUNTO DE ATRIBUTOS

En principio, podemos calcular el cierre S^+ de un conjunto dado *S* de DFs, mediante un algoritmo que dice: "aplicar repetidamente las reglas de la sección anterior hasta que dejen de producir nuevas DFs". En la práctica, existe poca necesidad de calcular el cierre *como tal* (lo que está bien, ya que el algoritmo que acabamos de mencionar difícilmente es eficiente). Sin embargo, en esta sección, mostraremos cómo calcular un cierto subconjunto del cierre: es decir, ese subconjunto que consiste en todas las DFs con un cierto conjunto Z (especificado) de atributos, como la parte izquierda. Para ser más precisos, mostraremos cómo a partir de una varrel *R,* un conjunto Z de atributos de *R* y un conjunto *S* de DFs que son válidas para *R,* podemos determinar el conjunto de todos los atributos de *R* que son dependientes funcionalmente de Z; el también llamado **cierre *Z** de Z bajo S.*** La figura 10.2 presenta un algoritmo sencillo para calcular ese cierre. *Ejercicio:* Demuestre que el algoritmo es correcto.

```
CIERRE[Z,S]  := Z
haz   "por siempre"
      para cada DF  X -> Y en S
          haz
              si X <  CIERRE[Z,S]          /* < = "subconjunto de" */
               entonces CIERRE[Z. S]  := CIERRE[Z.S] u Y;
          fin  :
      si CIERRE[Z,          5] no en esta iteración
      entonces sal  del ciclo  :          /* cálculo concluido       */
fin   •
```

Figura 10.2 Cálculo del cierre Z^+ de Z bajo *S.*

Ejemplo: Suponga que tenemos la varrel *R* con los atributos *A, B, C, D, E, F* y las DFs

A -» *BC*
E -> *CF*
B -> *E*
CD -> *EF*

*E1 conjunto de DFs con Z como la parte izquierda es el conjunto que consiste en todas las DFs de la forma Z —> Z'; donde Z' es algún subconjunto de Z^+. El cierre S^+ del conjunto original *S* es entonces la unión de todos estos conjuntos de DFs tomados sobre todos los posibles conjuntos de atributos Z.

Ahora calculamos el cierre $\{A,B\}^+$ del conjunto de atributos *{A,B}* bajo este conjunto de DFs.

1. Iniciamos el resultado CIERRE[Z,S] a *{A,B}.*
2. Ahora recorremos cuatro veces el ciclo interior, una para cada una de las DFs dadas. En la primera iteración (para la DF *A —> BC),* encontramos que la parte izquierda es ciertamente un subconjunto de CIERRE[Z,S] como está calculado hasta ahora; así que agregamos los atributos *(B* y) C al resultado. Ahora, CIERRE *[Z,S]* es el conjunto *{A,B,C}.*
3. En la segunda iteración (para la DF *E —¥ CF),* encontramos que la parte izquierda *no* es un subconjunto del resultado como está calculado hasta ahora; el cual por lo tanto, permanece sin cambio.
4. En la tercera iteración (para la DF *B —> E),* agregamos *E* a CIERRE[Z.S], que ahora tiene el valor *{A,B,C,E}.*
5. En la cuarta iteración (para la DF *CD —> EF), CIERRE[Z,S]* permanece sin cambio.
6. Ahora volvemos a recorrer cuatro veces el ciclo interior. En la primera iteración el resultado no cambia; en la segunda se amplía a *{A,B,C,E,F};* en la tercera y en la cuarta no cambia.
7. Ahora volvemos a recorrer cuatro veces el ciclo interior. CIERRE[Z,5] no cambia, por lo que todo el proceso termina con $\{A,B\}^+ = \{A,B,C,E,F\}$. |

Un corolario importante de lo anterior es el siguiente: dado un conjunto *S* de DFs, podemos saber fácilmente si una DF específica *X —> Y* se desprende de *S,* debido a que esa DF se desprenderá si y solamente si *Yes* un subconjunto del cierre X^+ de *X* bajo *S.* En otras palabras, ahora tenemos una forma sencilla de determinar si una DF dada *X —> Y* está en el cierre $5''^+$ de *S,* sin tener que calcular ese cierre S^+.

Otro corolario importante es el siguiente. Recuerde del capítulo 8 que una **superclave** de una varrel *R* es un conjunto de atributos de *R* que incluye como subconjunto alguna clave candidata de *R;* por supuesto, no necesariamente un subconjunto propio. Ahora bien, de la definición podemos deducir que las superclaves de una determinada varrel *R* son precisamente esos subconjuntos *K* de los atributos de *R* tales que la DF

es válida para todo atributo *A* de *R.* En otras palabras, *K* es una superclave si y sólo si el cierre K^+ de *K,* bajo el conjunto dado de DFs, es precisamente el conjunto de todos los atributos de *R* (y *K* es una clave *candidato* si y sólo si es una superclave irreducible).

10.6 CONJUNTOS DE DEPENDENCIAS IRREDUCIBLES

Sean *SI y S2* dos conjuntos de DFs. Si cada DF implicada por *SI* está implicada por *S2,* es decir, si Sl^+ es un subconjunto de $S2^+$, decimos que *S2* es una **cobertura** de *SI.** Esto significa que si el DBMS hace cumplir las DFs en *S2,* entonces estará haciendo cumplir automáticamente las DFs en *SI.*

*Algunos autores emplean el término "cobertura" para referirse a lo que nosotros llamaremos (en un momento más) un conjunto *equivalente.*

Después, si *S2* es una cobertura de *SI* y *SI* es una cobertura de 52, es decir si *SI* - S2**, decimos que *SI* y *S2* son equivalentes. Es claro que si *SI* y *S2* son equivalentes, entonces cuando el DBMS hace cumplir las DFs en 52 hará cumplir automáticamente las DFs en *SI* y *via*

Ahora bien, definimos un conjunto *S* de DFs como **irreducible*** si y sólo si satisface las siguientes tres propiedades:

1. La parte derecha (el dependiente) de toda DF en *S* involucra sólo un atributo (es decir, es un conjunto individual).
2. La parte izquierda (el determinante) de toda DF en *S* es a su vez irreducible, lo que significa que no es posible descartar ningún atributo del determinante sin cambiar el cierre *S** (es decir, sin convertir a *S* en algún conjunto no equivalente a *S).* Diremos que dicha DF es **irre ducible a la izquierda.**
3. No es posible descartar de *S* ninguna DF sin cambiar el cierre S^+ (es decir, sin convertir S en algún conjunto no equivalente a *S).*

Por cierto, con respecto a los puntos 2 y 3, observe cuidadosamente que no es necesario conocer exactamente cuál *es* el cierre 5^+ para saber si éste cambiaría al descartar algo. Por ejemplo, considere la conocida varrel de partes P. Las siguientes DFs (entre otras) son válidas para esa varrel:

```
P# -» PARTE
P# -+ COLOR
P# -> PESO
P# -> CIUDAD
```

Este conjunto de DFs se ve claramente como irreducible; en cada caso, la parte derecha es un solo atributo, la parte izquierda es a su vez obviamente irreducible y no es posible descartar ninguna de las DFs sin cambiar el cierre (es decir, sin *perder alguna información).* En contraste. los siguientes conjuntos de DFs no son irreducibles:

1. `p# -> { PARTE. COLOR }`
   `p# -» PESO`
   `p# -» CIUDAD`

   : Aquí, la parte derecha de la primera DF no es un conjunto individual.

2. { p#, `PARTE } -> COLOR`

   `p# -» PARTE`
   `p# -» PESO`
   `p# -» CIUDAD`

   Aquí, se puede simplificar la primera DF al eliminar PARTE de la parte izquierda sin modificar el cierre (es decir, no es irreducible hacia la izquierda).

3. p# -* P#

   `p# «♦ PARTE`
   `p# -» COLOR`
   `p# -» PESO`
   `p# -V CIUDAD`

   Aquí, es posible descartar la primera DF sin cambiar el cierre.

*Por lo regular denominado *mínimo* en la literatura.

Ahora afirmamos que para todo conjunto de DFs, existe por lo menos un conjunto equivalente que es irreducible. De hecho, esto es fácil de apreciar. Sea *S* el conjunto original de DFs. Gracias a la regla de descomposición, podemos dar por hecho —sin perder la generalidad— que toda DF en *S* tiene una parte derecha de un solo elemento. Luego, para cada DF/en *S,* examinamos cada atributo *A* en la parte izquierda de/- si la eliminación de *A* de la parte izquierda de /no tiene efecto en el cierre S^+, eliminaremos *A* de la parte izquierda de/ Entonces, para cada DF/que quede en *S,* si la eliminación de/de *S* no tiene efecto en el cierre S^+, eliminaremos/ de *S.* El conjunto final *S* es irreducible y es equivalente al conjunto original *S.*

Ejemplo: suponga que tenemos la varrel *R* con los atributos *A, B, C, D* y las DFs

```
A  -> BC
B  -> C
A  -» B
AB -» C
AC -> D
```

Ahora calculamos un conjunto irreducible de DFs que sea equivalente a este conjunto dado.

1. El primer paso es reescribir las DFs de modo que cada una tenga una parte derecha de un solo elemento:

```
A  -> 8
A  -> C
B  -i C
A  -» e
AB -» C
AC -> D
```

   De inmediato observamos que la DF *A* —> *B* ocurre dos veces, así que es posible eliminar una de las ocurrencias.

2. Después, podemos eliminar el atributo C de la parte izquierda de la DF *AC* —> *D,* ya que tenemos que *A* —> *C* (por lo tanto A —» *AC,* por aumento) y tenemos que AC —> D (por lo tanto *A* —> *D,* por transitividad); por lo tanto la C en la parte izquierda de *AC* —» Z) es re dundante.
3. Enseguida, observamos que la DF *AB* —> *C* puede ser eliminada, ya que otra vez tenemos que *A*—>*C* (por lo tanto *AB* —> *CB* por aumento) y por lo tanto *AB* —» C por descomposición.
4. Por último, la DF *A* —> C está implicada por las DFs A —> Z? y 5 —> C, de manera que tam bién puede ser eliminada. Lo que nos queda es:

```
A  -»  S
S  ->  C
4      ->
D
```

   Este conjunto es irreducible. |

Decimos que un conjunto / de DFs, que es irreducible y equivalente a algún otro conjunto *S* de DFs, es un **equivalente irreducible** de *S.* Por lo tanto, dado algún conjunto específico *S* de DFs que deba ser cumplido, basta con que el sistema encuentre y haga cumplir en su lugar las DFs de un equivalente irreducible 7 (y de nuevo, para calcular un equivalente irreducible 7, no hay necesidad de calcular el cierre S^+). Sin embargo, debemos dejar en claro que un conjunto dado de DFs no tiene necesariamente un equivalente irreducible *único* (vea el ejercicio 10.12).

10.7 RESUMEN

Una DF **(dependencia funcional)** es un vínculo muchos a uno entre dos conjuntos de atributos de una varrel determinada. Dada una varrel *R,* decimos que la DF *A* —»*B* (donde A y *B* son subconjuntos del conjunto de atributos de *R)* es válida para *R* si y solamente si siempre que dos tupias de *R* tienen el mismo valor de A, tienen también el mismo valor de *B.* Toda varrel satisface necesariamente ciertas DFs **triviales;** una DF es trivial si y sólo si la parte derecha (el **dependiente)** es un subconjunto de la parte izquierda (el **determinante).**

Ciertas DFs implican a otras. Dado un conjunto *S* de DFs, el **cierre** S^+ de ese conjunto es el conjunto de todas las DFs implicadas por las DFs en *S.* S^+ es necesariamente un superconjunto de *S.* Las **reglas de inferencia de Armstrong** proporcionan una base **firme y completa** para calcular S^+ a partir de *S* (sin embargo, por lo regular no hacemos ese cálculo en realidad). De las reglas de Armstrong podemos deducir otras reglas útiles.

Dado un subconjunto Z del conjunto de atributos de la varrel *R* y un conjunto *S* de DFs que es válido para *R,* el **cierre** Z^+ de Z bajo *S* es el conjunto de todos los atributos A de *R* tales que la DF Z —^ A es un miembro de S^+. Si Z^+ consiste en todos los atributos de *R,* se dice que Z es una **superclave de *R* (y** una **clave candid ata** es una superclave irreducible). Dimos un algoritmo sencillo para calcular Z^+ a partir de Z y 5; y por lo tanto, una forma sencilla para determinar s una DF dada *X* —> *Y* es un miembro de S^+ *(X* —» y es un miembro de S^+ si y sólo si *Y* es un subconjunto de X^+*).*

Dos conjuntos de DFs *SI y S2* son **equivalentes** si y solamente si son **coberturas** entre sí; es decir, si y sólo si $Sl^+ = S2^+$. Todo conjunto de DFs es equivalente a por lo menos un conjunto **irreducible.** Un conjunto de DFs es irreducible si (a) toda DF del conjunto tiene una parte derecha de un solo elemento, (b) ninguna DF del conjunto puede ser descartada sin cambiar el cierre del conjunto y (c) ningún atributo puede ser descartado de la parte izquierda de ninguna DF del conjunto, sin cambiar el cierre del mismo. Si / es un conjunto irreducible equivalente a *S,* el cumplimiento de las DFs en / hará cumplir automáticamente las DFs en *S.*

En conclusión, observamos que muchas de las ideas anteriores pueden ser extendidas, en general, a restricciones de integridad, no sólo a DFs. Por ejemplo, en general es cierto que:

- Ciertas restricciones son triviales;
- Ciertas restricciones implican otras;
- El conjunto de todas las restricciones implicadas por un determinado conjunto puede ser considerado como el cierre de dicho conjunto;
- La cuestión de si una restricción específica está en un cierto cierre —es decir, si la restric ción específica está implicada por ciertas restricciones dadas— es un problema práctico in teresante;
- La cuestión de encontrar un equivalente irreducible para un conjunto determinado de res tricciones es un problema práctico interesante.

Lo que hace que las DFs en particular sean mucho más manejables que las restricciones de integridad en general, es la existencia de un conjunto firme y completo de reglas de inferencia de DFs. Las secciones de "Referencias y bibliografía" de este capítulo y del capítulo 12, le ofrecen referencias a artículos que describen otras clases específicas de restricciones —DMVs, DJs, e DINs— para las cuales también existen dichas reglas de inferencia. Sin embargo, en este libro

decidimos no dar a esas otras clases de restricciones un tratamiento tan amplio y tan formal como el que dimos a las DFs.

EJERCICIOS

10.1 Sea *R* una varrel de grado n. ¿Cuál es el número máximo de dependencias funcionales (tanto triviales como no triviales) que *R* puede satisfacer?

10.2 ¿Qué significa decir que las reglas de inferencia de Armstrong son firmes? ¿Qué significa decir que son completas?

10.3 Demuestre las reglas de *reflexividad, aumento* y *transitividad.* Tome sólo la definición básica de dependencia funcional.

10.4 Demuestre que las tres reglas del ejercicio anterior implican a las reglas de *autodeterminación, descomposición, unión* y *composición.*

10.5 Demuestre el "teorema de unificación general" de Darwen. ¿Qué reglas utilizó de los dos ejer cicios anteriores? ¿Qué reglas pueden derivarse como casos especiales del teorema?

10.6 Defina (a) el cierre de un conjunto de DFs; (b) el cierre de un conjunto de atributos bajo un con junto de DFs.

10.7 Enumere todas las DFs satisfechas por la varrel de envíos VP.

10.8 La varrel *R{A,B,C,D,E,F,G)* satisface las siguientes DFs:

```
A  -> B
BC -» DE
AE -> G
```

Calcule el cierre $\{A,C\}^+$ bajo este conjunto de DFs. ¿La DF *ACF —> DG* está implicada por este conjunto?

10.9 ¿Qué significa decir que dos conjuntos *SI y S2* de DFs son equivalentes?

10.10 ¿Qué significa decir que un conjunto de DFs es irreducible?

10.11 Aquí tenemos dos conjuntos de DFs para la varrel *R{A,B,C,D,E}.* ¿Son equivalentes?

1. **A -> B AB ~> C D -> AC D -> f**
2. A -> BC D -> AE

10.12 La varrel *R{A,B,C,D,E,F}* satisface las siguientes DFs:

```
AB  -> C
C   -> A
BC  -> D
ACD -> B
BE  -> C
CE  -> FA
CF  -> BD
D   -> EF
```

Encuentre un equivalente irreducible para este conjunto de DFs.

10.13 Una varrel HORARIO es definida con los siguientes atributos:

D Día de la semana (1 a 5)
P Periodo dentro del día (1 a 8)
C Número de salón de clases

T Nombre del maestro
L Nombre de la materia

La tupia *{D:d,P:p,C:c,T:t,L:l}* aparece en esta varrel si y sólo si en el momento *{D:d,P:p}* la materia / es impartida por el maestro *t* en el salón *c.* Podemos dar por hecho que las materias tienen una duración de un periodo y que cada materia tiene un nombre que es único con respecto a las demás que se imparten en la semana. ¿Qué dependencias funcionales son válidas en esta varrel? ¿Cuáles son las claves candidatas?

10.14 Una varrel NDIR es definida con los atributos NOMBRE (único), CALLE, CIUDAD, ESTADO y CP. Para cualquier Código Postal (CP) dado, sólo existe una ciudad y un estado. También, para cualquier calle, ciudad y estado dados, sólo existe un Código Postal. Proponga un conjunto irreducible de DFs para esta varrel. ¿Cuáles son las claves candidatas?

10.15 La varrel *R[A,B,C,D,E,F,G,H,I,J}* satisface las siguientes DFs:

AB -> E
AB -> G
B -> F
C -» J
CJ -i I
G -» H

¿Es éste un conjunto irreducible? ¿Cuáles son las claves candidatas?

REFERENCIAS Y BIBLIOGRAFÍA

10.1 W. W. Armstrong: "Dependency Structures of Data Base Relationships", Proc. IF1P Congress. Estocolmo, Suecia (1974).

El artículo que formalizó primero la teoría de las DFs (es el origen de los "Axiomas de Armstrong"). El artículo ofrece también una caracterización precisa de las claves candidatas.

10.2 Marco A. Casanova, Ronald Fagin y Christos H. Papadimitriou: "Inclusion Dependencies and Their Interaction with Functional Dependencies", Proc. 1 st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, Calif, (marzo, 1982).

■ Las **dependencias de inclusión** (DINs) pueden ser consideradas como una generalización de las restricciones referenciales. Por ejemplo, la DIN

VP.V# -> V.V#

(no es la notación que emplea el artículo) establece que el conjunto de valores que aparecen en el atributo V# de la varrel VP debe ser un subconjunto del conjunto de valores que aparece en el atributo V# de la varrel V. Por supuesto, este ejemplo en particular *es* de hecho una restricción referencial; sin embargo, en general no hay un requerimiento de que la parte izquierda de una DIN sea una clave externa o la parte derecha una clave candidata. *Nota:* Las DINs sí tienen algunos puntos en común con las DFs, ya que ambas representan vínculos muchos a uno; sin embargo, las DINs por lo regular abarcan a varias varrels mientras que las DFs no.

El artículo proporciona un conjunto firme y completo de reglas de inferencia para DINs, las cuales podrían ser enunciadas (en general), como sigue: L A -»- A.
2. Si AB -► CD, entonces A -► C y B -► D.
3. Si A -► B y B -► C, entonces A -►■ C.

10.3 R. G. Casey y C. Delobel: "Decomposition of a Data Base and the Theory of Boolean Switching Functions", *IBM J. R&D 17,* No. 5 (septiembre, 1973).

Este artículo muestra que para cualquier varrel dada, el conjunto de DFs (denominado *relaciones funcionales*, en este documento) satisfecho por dicha varrel puede ser representado mediante una "función de conmutación lógica". Es más, la función es única en el siguiente sentido: las DFs originales pueden ser especificadas en muchas formas superficialmente distintas (aunque de hecho equivalentes), lo que da lugar cada una a una función lógica superficialmente diferente; pero todas esas funciones pueden ser reducidas —mediante las leyes del álgebra de Boole— a la misma forma canónica. El problema de *descomponer* la varrel original (es decir, en una forma sin pérdida; vea el capítulo 11) se muestra entonces como equivalente lógicamente al bien entendido problema del álgebra de Boole de encontrar "un conjunto de cobertura de implicantes primos" para la función lógica correspondiente a esa varrel junto con sus DFs. De ahí que el problema original pueda ser transformado en uno equivalente en el álgebra de Boole, y sea posible emplear técnicas bien conocidas para enfrentarlo.

Este artículo fue el primero de varios para establecer paralelos entre la teoría de la dependencia y otras disciplinas: Por ejemplo, vea más adelante la referencia [10.8], además de varias de las referencias del capítulo 12.

10.4 E. F. Codd: "Further Normalization of the Data Base Relational Model", en Randall J. Rustin (ed.), *Data Base Systems, Courant Computer Science Symposia Series 6.* Englewood Cliffs, N.J.: Prentice-Hall (1972).

El artículo que presentó por vez primera el concepto de dependencia funcional (además de un memorando interno anterior de IBM). La "normalización adicional" (que explicamos en el capítulo 11) del título se refiere a la disciplina específica del diseño de bases de datos; la finalidad del artículo fue mostrar, de manera muy específica, la aplicabilidad de las ideas de la dependencia funcional para el problema del diseño de bases de datos. (De hecho, las DFs representaron el primer ataque científico a ese problema.) Sin embargo, desde entonces la dependencia funcional ha mostrado ser en sí misma de una aplicabilidad mucho más amplia.

10.5 E. F. Codd: "Normalized Data Base Structure: A Brief Tutorial", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif, (noviembre, 1971).

Una introducción tutorial a las ideas de la referencia [10.4].

10.6 Hugh Darwen: "The Role of Functional Dependence in Query Decomposition", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).

Este artículo proporciona un conjunto de reglas de inferencia mediante las cuales es posible inferir las DFs válidas para una varrel derivada cualquiera a partir de aquellas que son válidas para las varrels de las que la varrel en cuestión se deriva. El conjunto de DFs así inferido puede ser inspeccionado después para determinar las claves candidatas de la varrel derivada, proporcionando de esta manera las reglas de inferencia de *claves candidatas* que mencionamos brevemente en los capítulos 8 y 9. El artículo presenta la forma en que estas reglas diversas pueden ser utilizadas para ofrecer mejoras significativas en el rendimiento, funcionalidad y utilización del DBMS.

10.7 Hugh Darwen: "OObservations *[sic]* of a Relational Bigot", presentación para BCS Special In terest Group on Formal Aspects of Computing Science, Londres, Reino Unido (diciembre 21, 1990).

10.8 R. Fagin: "Functional Dependencies in a Relational Database and Propositional Logic", *IBM J. R&D 21,* No. 6 (noviembre, 1977).

Muestra que los "Axiomas de Armstrong" [10.1] son estrictamente equivalentes al sistema de enunciados de implicación de la lógica de proposiciones. En otras palabras, el documento define una transformación entre las DFs y los enunciados de proposiciones, y luego muestra que una determinada DF/es una consecuencia de un conjunto dado *S* de DFs, si y sólo si la proposición correspondiente a/es una consecuencia lógica del conjunto de proposiciones correspondiente a *S.*

10.9 Claudio L. Lucchesi y Sylvia L. Osborn: "Candidate Keys for Relations", *J. Comp. and Sys. Sci ences 17,* No. 2 (1978).

Presenta un algoritmo para encontrar todas las claves candidatas de una varrel determinada, c el conjunto de DFs válidas de dicha varrel.

RESPUESTAS A EJERCICIOS SELECCIONADOS

10.1 Una DF es básicamente un enunciado de la forma A —> *B* en donde *A y B* son cada uno subconjuntos del conjunto de atributos de *R.* Puesto que un conjunto de *n* elementos tiene 2" subconjuntos posibles, tanto A como *B* tienen 2" valores posibles y por lo tanto, un límite superior del número de DFs posibles es $2^{2"}$. 10.5

1. *A* -> s	(dado)
2. *c* -> *o*	(dado)
3. *A* -> s n *c*	(dependencia de junta, 1)
4. *c* - *B* -> c - B	(autodeterminación)
5. / \ u (c - s) - > (B n C) u (C - s)	(composición, 3, 4)
6. A u (c - B) - > c	(simplificación, 5)
7 . A U (C - B) - > D	(transitividad, 6, 2)
8.A u (c - s) - > S u D	(composición, 1, 7)

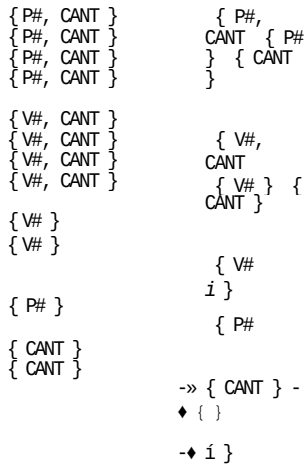
Esto completa la prueba.

Las reglas utilizadas en la prueba son las que se indican en los comentarios anteriores. Los siguientes son casos especiales del teorema de Darwen: unión, transitividad, composición y aumento. Así también es la siguiente regla útil:

■ Si A -> *B* y *AB* ->C, entonces A ->*C.* 10.7 El conjunto completo de DFs —es decir el cierre para la varrel VP— es como sigue:

```
{ V#, P#, CANT } -^ { V#, P#, CANT }
{ V#, P#, CANT } -> { V#, P# }
{ V#, P#, CANT } -> { P#, CANT }
{ V#, P#, CANT } -> { V#, CANT }
{ V#, P#, CANT } -> { V# }
{ V#, P#, CANT } -> { P# }
{ V#, P#, CANT } -> { CANT }
{ V#, P#, CANT } -> { }

{ V#, P# }       -+ { V#, P#, CANT }
{ V#, P# }       -» { V#, P# }
{ V#, P# }       -> { P#, CANT }
{ V#, P# }       -> { V#, CANT }
{ V#, P# }       -» { V# }
{ V#, P# }       -> { P# }
{ V#, P# }       -^ { CANT }
{ V#, P# }       -> { }
```



10.8 $\{A,C\}^+ = \{A,B,C,D,E\}$. La respuesta a la segunda parte de la pregunta es sí. 10.11
Son equivalentes. Enumeremos las DFs del primer conjunto como sigue:

1. **A -> B**
2. **AB -> C**
3. **D -> AC**
4. **D -» B**

Primero, podemos reemplazar a 3 por:

A continuación, 1 y 2 juntos implican que 2 puede ser reemplazado por:

2. **A -> C**

Pero ahora tenemos *D* —> *A* y *A* —>C, así que *D* —>*C* es implicada (por transitividad) y por ello puede ser eliminada, quedando:

3. **0 -+ A**

Por lo tanto, el primer conjunto de DFs es equivalente al siguiente conjunto irreducible:

A -> e
A -» C
O -» /I
D -» E

El segundo conjunto dado de DFs

A -> SC

es claramente equivalente a este conjunto irreducible. Por lo tanto, los dos conjuntos dados son equivalentes.

10.12 El primer paso es reescribir el conjunto dado de tal forma que toda DF tenga una parte derecha de un solo elemento:

1 *AB* -» C
2 *C* -» *A*
3 *BC* -» D
4. *ACD* -> e
5 *BE* -> c
6. *CE* -» *A*
7. *CE* -» *F*
8. *CF* -> e
9. *CF* -• D
1 *D* -» E
1 *D* -> *F*

Ahora:

- 2 implica a 6, así que podemos eliminar a 6.
- 8 implica a CF —>*BC* (por aumento), la cual con 3 implica a *CF* —> D (por transitividad), así que podemos eliminar a 10.
- 8 implica a ACF —» *AB* (por aumento), 11 implica a ACD —> ACF (por aumento) y también a *ACD* —> AB (por transitividad) y también a ACD —> *B* (por descomposición); de manera que podemos eliminar a 4.

No son posibles más reducciones, así que nos queda el siguiente conjunto irreducible:

AB
C -> A
BC -» D
BE -> C
CE H» F
CF -> B
D 4 H
D -> F

Como

- 2 implica a *CD* —» ACD (por composición), la cual con 4 implica a *CD* —> £ (por transitividad); así que podemos reemplazar a 4 por CD —> B.
- 2 implica a 6; así que (al igual que antes) podemos eliminar a 6.
- 2 y 10 implican a *CF* —> AD (por composición), que implica a CF —> *ADC* (por aumento), que con el 4 (original) implica a *CF* -» B (por transitividad); así que podemos eliminar a 8.

No son posibles más reducciones, así que nos queda el siguiente conjunto irreducible:

A -t C
C -» *A*
B -t 0
C -» S
B -» C
C ->
CF -4 0
D -♦ £
D -» *F*

Por lo tanto, observe que hay dos distintos equivalentes irreducibles para este conjunto original de DFs.

10.13 Las claves candidatas son *L, DPC y DPT.*

10.14 Al abreviar NOMBRE, CALLE, CIUDAD, ESTADO y CP como *N, R, C,TyZ,* respectiva mente, tenemos:

N -> *RCT* *RCT* -> Z Z -» *CT*

Un conjunto irreducible equivalente obvio es:

« -> fl « -> C *N* -» 7" fid - > Z Z -> C Z -> *T*

La única clave candidata es ¿V.

10.15 No damos una respuesta completa a este ejercicio, pero nos contentamos con las siguientes ob servaciones. Primero, es claro que el conjunto no es irreducible, ya que C —> *J* y *CJ* —> /juntos im plican a *C* —> /. Segundo, una *superclave* obvia es {A,5,C,D,G,7}; es decir, el conjunto de todos los atributos mencionados en la parte izquierda de las DFs dadas. Podemos eliminar *J* de este conjunto debido a que *C* —> *J y* podemos eliminar *G* porque *AB* —> *G.* Puesto que *A, B, C, D* no aparecen en la parte derecha de ninguna de las DFs dadas, podemos inferir que *{A,B,C,D}* es una clave candidata.

CAPITULO 11

Normalización adicional I: 1FN, 2FN, 3FN, FNBC

11.1 INTRODUCCIÓN

A lo largo de este libro hemos hecho uso de la base de datos de proveedores y partes como un ejemplo continuo, con un diseño lógico (esbozado) como sigue:

```
V   { V#, PROVEEDOR, STATUS, CIUDAD }
    PRIMARY KEY { V# }

P  { P#, PARTE, COLOR, PESO, CIUDAD }
    PRIMARY KEY { P# }

VP  { V#, P#, CANT }
    PRIMARY KEY { V#, P# }
    FOREIGN KEY {VI } REFERENCES V
    FOREIGN KEY { P# } REFERENCES P
```

Ahora bien, este diseño nos hace pensar que todo él está correcto: es "obvio" que las tres varrels V, P y VP son necesarias; también es "obvio" que el atributo de proveedores CIUDAD pertenece a la varrel V, el atributo de partes COLOR pertenece a la varrel P, el atributo de envíos CANT pertenece a la varrel VP y así sucesivamente. Pero ¿qué es lo que nos dice que estas cosas son así? Podemos obtener alguna respuesta a esta pregunta si observamos lo que sucede al cambiar el diseño en alguna forma. Por ejemplo, suponga que el atributo de proveedores CIUDAD es sacado de la varrel de proveedores e insertado en la de envíos (intuitivamente en el lugar equivocado, ya que la "ciudad del proveedor" le concierne obviamente a los proveedores, no a los envíos). La figura **11.1** de la siguiente página —una variante de la figura 10.1 del capítulo 10— presenta un valor de ejemplo para esta varrel de envíos modificada. *Nota:* Con el fin de evitar confusiones con nuestra varrel usual de envíos, VP, nos referiremos a esta versión revisada como VCP (tal como lo hicimos en el capítulo 10).

Basta con una mirada a la figura para ver lo que está mal en el diseño: la **redundancia.** Para ser específicos, cada tupia de VCP para el proveedor VI nos dice que VI está ubicado en Londres, cada tupia del proveedor V2 nos dice que V2 está ubicado en París, etcétera. De manera más general, el hecho de que un proveedor determinado esté ubicado en una ciudad dada es enunciado tantas veces como los envíos que hay de ese proveedor. A su vez, esta redundancia conduce a varios problemas adicionales. Por ejemplo, después de una actualización, el proveedor VI podría aparecer como si estuviera ubicado en Londres, de acuerdo con una tupia, y

VCP

V#	CIUDAD	P#	CANT
V1	Londres	P1	300
V1	Londres	P2	200
V1	Londres	P3	400
V1	Londres	P4	200
V1	Londres	P5	100
V1	Londres	P6	100
V2	París	P1	300
V2	París	P2	400
V3	París	P2	200
V4	Londres	P2	200
V4	Londres	P4	300
V4	Londres	P5	400

Figura **11.1** Valor de ejemplo de la varrel VCP.

ubicado en Amsterdam, de acuerdo con otra.* De modo que tal vez un buen principio de diseño sea "cada hecho en un lugar" (es decir, evitar la redundancia). *El tema de la normalización adicional es en esencia sólo una formalización de ideas sencillas como ésta,* pero una formalización que sí tiene una aplicación muy práctica para el problema del diseño de bases de datos.

Desde luego (como vimos en el capítulo 5), por lo que respecta al modelo relacional, las *relaciones* siempre están normalizadas. En cuanto a las *varrels*, podemos decir que también están normalizadas (siempre y cuando sus valores válidos sean relaciones normalizadas); por lo tanto, en lo que respecta al modelo relacional, también las varrels están siempre normalizadas. En forma equivalente, podemos decir que las varrels (y las relaciones) están siempre en la **primera forma normal** (abreviada 1FN). En otras palabras, "normalizada" y "1FN" significan *exactamente lo mismo;* aunque debe tener presente que el término "normalizado" se usa a menudo para indicar uno de los niveles más altos de normalización (por lo regular a la *tercera* forma normal, 3FN). Este último uso es descuidado pero muy común.

Ahora bien, una varrel dada podría normalizarse en el sentido anterior y poseer aún ciertas propiedades indeseables. La varrel VCP es uno de estos casos (vea la figura 11.1). Los principios de la normalización adicional nos permiten reconocer dichos casos y reemplazar esas varrels por otras que sean en cierto modo más atractivas. Por ejemplo, en el caso de la varrel VCP, dichos principios nos diñan precisamente lo que está mal con esa varrel y nos dirían cómo reemplazarla por varrels "más atractivas": una con el encabezado {V#,CIUDAD} y otra con el encabezado {V#,P#,CANT}.

---

*A lo largo de este capítulo y el siguiente, es necesario tener presente (¡de manera suficientemente realista!) que los predicados de varrel no se hacen cumplir en su totalidad; ya que si así fuera, no habría posibilidad de que surgieran problemas como éste (no sería posible actualizar la ciudad del proveedor VI en algunas tupias y en otras no). De hecho, una forma de pensar en la disciplina de la normalización es la siguiente: la normalización nos ayuda a estructurar la base de datos de tal forma que las actualizaciones de tupias individuales sean más aceptables lógicamente de lo que serían de otro modo (es decir, si el diseño no estuviese totalmente normalizado). Esta meta se logra debido a que los predicados de varrel son más sencillos con un diseño completamente normalizado.

Formas normales

El proceso de normalización adicional —a partir de aquí abreviado solamente *normalizada* se genera alrededor del concepto de las **formas normales.** Decimos que una varrel estáe forma normal en particular si satisface cierto conjunto de condiciones preestablecidas. Porej pío, decimos que una varrel está en la **segunda forma normal** (2FN) si y sólo si está en 1FN; además satisface otra condición, que abordaremos en la sección 11.3.

Se han definido muchas formas normales (vea la figura 11.2). Las tres primeras 2FN, 3FN) fueron definidas por Codd en la referencia [10.4], Como sugiere la figura todas las varrels normalizadas están en 1FN; algunas varrels 1FN están también en 2FN: \ algunas varrels 2FN están también en 3FN. El motivo de las definiciones de Codd fue 2FN era "más atractiva" que 1FN (en un sentido que explicaremos) y a su vez, 3FN erai atractiva que 2FN. Por lo tanto, el diseñador de bases de datos debe por lo general api hacia un diseño que comprenda varrels en 3FN, no hacia aquellas que estén simplemen 2FN o 1FN.

La referencia [10.4] también presentó la idea de un procedimiento (el llamado proced **miento de normalización)** mediante el cual una varrel que estuviera en alguna forma ni dada, digamos 2FN, podría ser reemplazada por un conjunto <fe varrels en alguna forma i atractiva, digamos 3FN. (Por supuesto, el procedimiento, tal como se definió originalmer sólo llegó hasta 3FN, pero posteriormente se extendió —como veremos en el siguiente cap tulo— hasta llegar a 5FN.) Podemos caracterizar a este procedimiento como *la reducción* si *siva de una colección dada de varrels hacia alguna forma más atractiva.* Observe que procedimiento es **reversible;** es decir, siempre es posible tomar los resultados del procedimie (digamos, el conjunto de varrels 3FN) y transformarlos de nuevo a su origen (digamos,! varrels 2FN originales). Desde luego, la reversibilidad es importante, ya que significaqu proceso de normalización *conserva la información.*

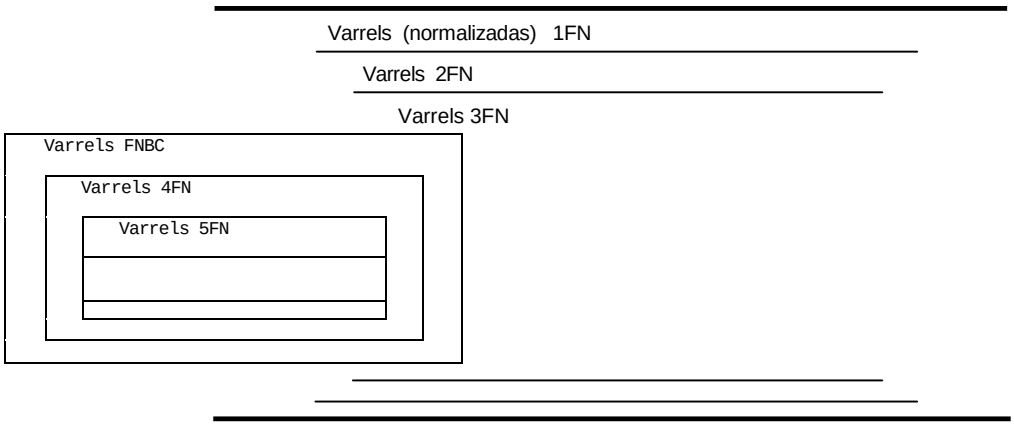


Figura 11.2 Niveles de normalización.

Para retomar el tema de las formas normales *en sí:* la definición original de Codd de 3FN [10.4] resultó padecer de ciertas inconsistencias, como veremos en la sección 11.5. Una versión modificada y más sólida, debida a Boyce y Codd, se dio en la referencia **[11.2];** más sólida en el sentido de que cualquier varrel que estuviera en 3FN por la nueva definición, estaba en realidad en 3FN por la antigua; aunque una varrel podría estar en 3FN por la definición anterior y no por la nueva. Por lo regular, a la nueva 3FN se le conoce como **Forma Normal de Boyce/Codd** (FNBC) a fin de distinguirla de la forma anterior.

Posteriormente, Fagin [11.8] definió una nueva **"cuarta" forma normal** (4FN, "cuarta" porque en ese momento a la FNBC aún se le llamaba "tercera"). Y en la referencia [11.9], Fagin definió otra forma normal más a la que llamó **forma normal de proyección-junta** (FN/PJ, también conocida como **"quinta"** forma normal o 5FN). Como muestra la figura 11.2, algunas varrels FNBC están también en 4FN y algunas varrels 4FN están en 5FN.

A estas alturas bien puede estarse preguntando si existe un fin para esta progresión, y si habrá una 6FN, una 7FN y así *indefinidamente.* Aunque ésta es una buena pregunta, es obvio que aún no estamos en posición de considerarla en detalle. Nos contentamos con la declaración más bien equívoca de que de hecho hay formas normales adicionales que no muestra la figura 11.2, aunque la 5FN es la forma normal "final" en un sentido especial (pero importante). Regresaremos a esta cuestión en el capítulo 12.

Estructura del capítulo

El objetivo de este capítulo consiste en examinar los conceptos de la normalización adicional inclusive hasta la forma normal de Boyce/Codd (dejamos las otras dos para el capítulo 12). El plan del capítulo es como sigue. Después de esta introducción más bien extensa, la sección 11.2 explica los conceptos básicos de la **descomposición sin pérdida** y demuestra la importancia crucial de la **dependencia funcional** para este concepto (de hecho, la dependencia funcional forma la base de las tres formas normales originales de Codd, así como de la FNBC). Después, la sección 11.3 describe las tres formas normales originales y muestra por medio de ejemplos cómo puede llevarse una determinada varrel hasta 3FN, a través del proceso de normalización. La sección 11.4 se desvía ligeramente para considerar la cuestión de las **descomposiciones alternativas;** es decir, la cuestión de seleccionar la "mejor" descomposición de una varrel dada, cuando existe la opción. A continuación, la sección 11.5 explica la FNBC. Por último, la sección 11.6 presenta un resumen y ofrece algunas observaciones de conclusión.

Le advertimos que en lo que sigue no pretendemos ser rigoristas; más bien dependemos en gran medida de la mera intuición. De hecho, parte del punto es que los conceptos como descomposición sin pérdida, FNBC, etcétera —a pesar de la terminología un tanto esotérica— son ideas esencialmente muy sencillas y de sentido común. La mayoría de las referencias tratan el material en una manera mucho más formal y rigurosa. Usted puede encontrar un buen tutorial en la referencia [11.5].

Dos últimas observaciones de introducción:

1. Como ya sugerimos, la idea general de la normalización es que el diseñador de bases de datos debe enfocarse en las varrels en la "última" forma normal (5FN). Sin embargo, esta recomendación no debe ser interpretada como una ley; ocasionalmente (¡pero *muy* ocasionalmente!) podría haber buenas razones para descartar los principios de la normalización (por ejemplo, vea el ejercicio 11.7 al final del capítulo). De hecho, éste es un lugar tan bueno

como cualquier otro para señalar que el diseño de bases de datos puede ser una tarea en extremo compleja (por lo menos en el entorno de las "grandes bases de datos"; el diseño de bases de datos "pequeñas" es, por lo regular, comparativamente directo). La normalización es una ayuda útil en el proceso, pero no es una panacea; por lo tanto, aconsejamos a todo diseñador de una base de datos familiarizarse con los principios de la normalización, aunque no pretendemos decir que el diseño deba basarse necesariamente sólo en dichos principios. El capítulo 13 explica otros aspectos del diseño que tienen poco o nada que ver con la normalización como tal.

2. Como indicamos anteriormente, usaremos el procedimiento de normalización como base para introducir y explicar las distintas formas normales. Sin embargo, no pretendemos sugerir que en la práctica el diseño de base de datos en realidad se lleve a cabo aplicando ese procedimiento. De hecho, es probable que no sea así; es mucho más probable que en su lugar se use algún esquema de arriba hacia abajo como el que describimos en el capítulo 13. Entonces, las ideas de la normalización pueden ser empleadas para *verificar* que el diseño resultante no viole de manera no intencional cualquiera de los principios de la normalización. No obstante, el procedimiento de normalización sí proporciona una infraestructura conveniente a partir de la cual podemos describir esos principios. Por lo tanto, para los fines de este capítulo adoptamos la ficción útil de que en realidad estamos llevando a cabo el proceso de diseño mediante la aplicación de ese procedimiento.

11.2 LA DESCOMPOSICIÓN SIN PERDIDA Y LAS DEPENDENCIAS FUNCIONALES

Antes de entrar en los detalles del procedimiento de normalización, necesitamos examinar más de cerca un aspecto crucial de ese procedimiento; es decir, el concepto de **descomposición** sjn **pérdida.** Ya vimos que el procedimiento de descomposición implica separar o *descomponer* una determinada varrel en otras y además requiere que la descomposición sea *reversible,* de tal forma que no se pierda información en el proceso; en otras palabras, las únicas descomposiciones en las que estamos interesados son aquellas que de hecho son sin pérdida. Como veremos, la cuestión de si una descomposición dada es sin pérdida, está íntimamente ligada con el concepto de la dependencia funcional.

A manera de ejemplo, considere la conocida varrel de proveedores V con el encabezado {V#, STATUS, CIUDAD} (por razones de simplicidad, ignoramos PROVEEDOR). La figura 11.3 de la siguiente página presenta un valor de ejemplo para esta varrel y en las partas déla figura etiquetadas como (a) y (b), presenta dos posibles descomposiciones correspondientes a ese valor de ejemplo.

Al examinar las dos descomposiciones, observamos que:

1. En el caso (a) no se pierde información; los valores VST y VC aún nos indican que el pro veedor V3 tiene el status 30 y la ciudad París, y que el proveedor V5 tiene el status 30 y la ciudad Atenas. En otras palabras, la primera descomposición es ciertamente sin pérdida.
2. En contraste, en el caso (b) se pierde definitivamente información; aún podemos saber que ambos proveedores tienen el status 30, pero no podemos saber qué proveedor tiene cuál ciu dad. En otras palabras, la segunda descomposición no es sin pérdida sino **con pérdida.**

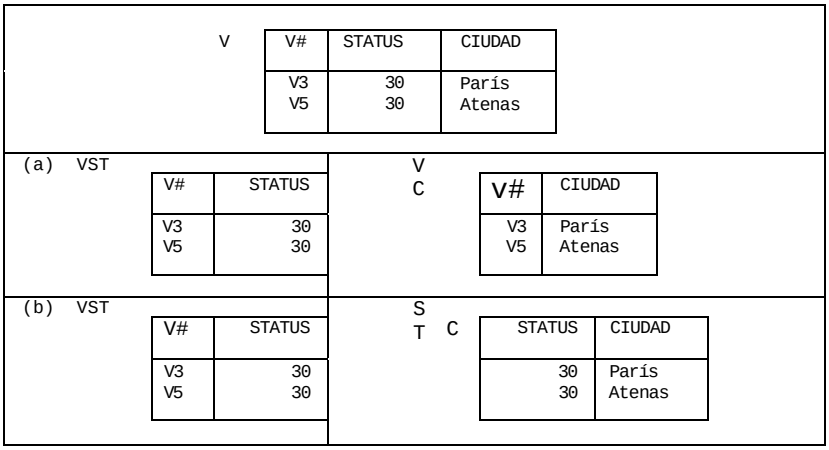


Figura 11.3 Valores de ejemplo para la varrel V y dos descomposiciones correspondientes.

¿Exactamente qué pasa aquí que hace que la primera descomposición sea sin pérdida y la otra con pérdida? Pues bien, primero observe que el proceso al que nos hemos estado refiriendo como "descomposición" es en realidad un proceso de **proyección;** en la figura, VST, VC y STC son proyecciones de la varrel original V. Así que el operador de descomposición en el procedimiento de normalización es de hecho el de *proyección. Nota:* En la parte II de este libro, con frecuencia decimos cosas como "VST es una proyección de la varrel V" cuando en forma más correcta deberíamos decir "VST es una varrel cuyo valor en cualquier momento dado es una proyección de la relación que es el valor de la varrel V en ese momento". Esperamos que esta forma de abreviar no genere confusión alguna.

Después, note que cuando en el caso (a) decimos que no se perdió información, lo que en realidad queremos decir es que *si juntamos de nuevo VST y VC, regresamos a la V original.* En contraste, si juntamos de nuevo VST y STC en el caso (b), *no* regresamos a la V original y por lo tanto hemos perdido información.* En otras palabras, la "reversibilidad" significa precisamente que *la varrel original es igual a la junta de sus proyecciones.* De ahí que, así como proyección es el operador de descomposición para la normalización, entonces el operador de recomposición es el de **junta.**

Pero la pregunta interesante es ésta: si *Rl* y *R2* son proyecciones de cierta varrel *R,* y *Rl* y *R2* incluyen entre sí todos los atributos de *R,* ¿qué condiciones deben satisfacerse para poder garantizar que al juntar de nuevo *Rl y R2* nos lleve de vuelta a la *R* original? Y es aquí donde entran las dependencias funcionales. Para regresar a nuestro ejemplo, observe que la varrel V satisface el conjunto irreducible de DFs

V# V# STATUS
CIUDAD

---

*Para ser más precisos, obtenemos de nuevo todas las tupias de la V original, junto con algunas tupias "falsas" adicionales; nunca podemos obtener *menos* que la V original. (*Ejercicio:* demuestre esta declaración.) Como en general no tenemos forma de saber cuáles tupias son falsas y cuáles genuinas, por lo tanto hemos perdido información.

Dado el hecho de que satisface estas DFs, seguramente no puede ser una coincidencia que la varrel V sea igual a la junta de sus proyecciones sobre {V#,STATUS) y {V#,CIUDAD). por supuesto, no lo es. De hecho, tenemos el siguiente *teorema* (debido a Heath **[11.4]):**

- **Teorema de Heath:** Sea *R{A,B,C}* una varrel (donde A, *B* y *C* son conjuntos de atributos! Si *R* satisface la DF A -> *B*, entonces *R* es igual a la junta de sus proyecciones sobre (A,fl) *y{A,C}.*

Si tomamos A como V#, *B* como STATUS y *C* como CIUDAD, este teorema confirma lo que ya observamos; es decir, que la varrel V puede descomponerse sin pérdida en sus proyección sobre {V#,STATUS} y {V#,CIUDAD}.

Al mismo tiempo, sabemos también que la varrel V *no puede* descomponerse sin pérdida en sus proyecciones sobre {V#,STATUS} y {STATUS,CIUDAD}. El teorema de Heatht explica por qué esto es así;* sin embargo, de manera intuitiva podemos ver que el problema es que *en la última descomposición se pierde una de las DFs.* En particular, la DF V# -»STATUS aún está representada (por la proyección sobre (V#,STATUS¡), pero se perdió laDFV#-CIUDAD.

Más sobre las dependencias funcionales

Concluimos esta sección con algunas observaciones adicionales concernientes a las DFs.

1. **DFs irreducibles a la izquierda:** Recuerde (del capítulo 10) que decimos que una DF es *irreducible a la izquierda* cuando su parte izquierda "no es demasiado grande". Por ejemplo, considere una vez más la varrel VCP de la sección 11.1. Esa varrel satisface la DF

   ```
   { V#, P# } -» CIUDAD
   ```

   Sin embargo, aquí el atributo P# de la parte izquierda es redundante para fines de dependencia funcional; es decir, también tenemos la DF

   ```
   V# -> CIUDAD
   ```

   (es decir, CIUDAD también es dependiente funcionalmente sólo de V#). Esta última DFe irreducible a la izquierda, pero la anterior no lo es; de manera equivalente, CIUDAD es*-*pendiente irreduciblemente* de V#, pero no dependiente irreduciblemente de {V#,P#).'

   Las DFs irreducibles a la izquierda y las dependencias irreducibles resultarán a su va importantes en la definición de la segunda y tercera formas normales (vea la sección 11.3).

---

*No lo hace porque es de la forma "si ... entonces ...", no de la forma "si *y sólo* si … entonces..."(i ejercicio 11.1 al final del capítulo). En el siguiente capítulo explicaremos una forma más sólida del ti de Heath (en la sección 12.2). [f]"DF irreducible a la izquierda" y "dependiente irreduciblemente" son nuestros términos preferidos **paral** que en la literatura se denomina regularmente como "DF **completa"** y "dependiente **totalmente"** (y asiles llamamos en las primeras ediciones de este **libro).** Estos últimos términos tienen la virtud de ser breves aunque menos descriptivos y menos aptos.

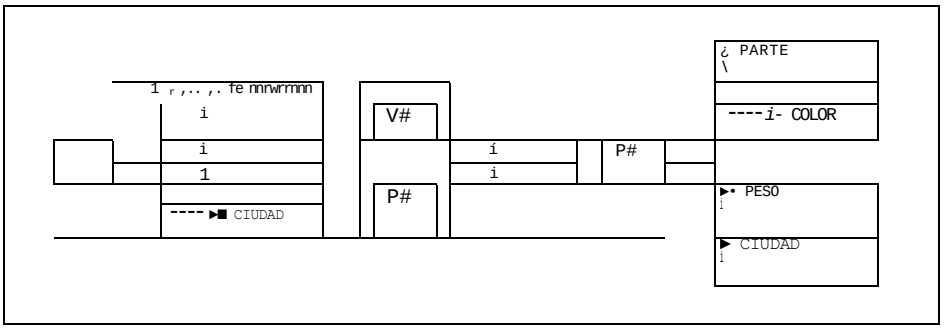


Figura 11.4 Diagramas DF para las varrels V, VP y P.

2. **Diagramas DF:** sea *R* una varrel y sea / algún conjunto irreducible de DFs que se aplica **a** *R* (de nuevo, consulte el capítulo 10 si necesita refrescar su memoria con respecto a los con juntos irreducibles de DFs). Es conveniente representar el conjunto / mediante un *diagrama de dependencia funcional* (diagrama DF). En la figura 11.4 presentamos los diagramas DF para las varrels V, VP y P, los cuales deben ser muy claros. En lo que resta del capítulo hare mos un amplio uso de dichos diagramas.

   Ahora bien, notará que cada flecha de la figura 11.4 es una *flecha que parte de una clave candidata* (en realidad, la clave primaria) de la varrel relevante. Por definición, siempre habrá flechas que partan de cada clave candidata.* Debido a que, para cada valor de una clave candidata, siempre existe un valor de todo lo demás, dichas flechas nunca podrán ser eliminadas. *Las dificultades surgen cuando existen otras flechas.* Por lo tanto, el proceso de normalización puede ser caracterizado (muy informalmente) como un procedimiento de eliminar flechas que no parten de claves candidatas.

3. **Las DFs son una noción semántica:** por supuesto, las DFs son una clase especial de res tricción de integridad y como tales, son definitivamente una noción *semántica.* El reconoci miento de las DFs es parte del proceso de entender lo que *significan* los datos; por ejemplo, el hecho de que la varrel V satisfaga la DF V# —> CIUDAD, significa que cada proveedor está ubicado precisamente en una ciudad. Para verlo de otra forma:
   - En la realidad, existe una restricción que la base de datos representa; es decir, que cada proveedor está ubicado precisamente en una ciudad;
   - Puesto que es parte de la semántica de la situación, esta restricción debe ser observada de alguna manera en la base de datos;
   - La forma de asegurar que sea observada consiste en especificarla dentro de la definición de la base de datos, de modo que el DBMS pueda hacerla cumplir;
   - La forma de especificarla dentro de la definición de la base de datos es mediante la de claración de la DF.

---

*Para ser más precisos, siempre habrá flechas que partan de las *superclaves.* Sin embargo, si el conjunto de DFs es irreducible como señalamos, todas las DFs (o "flechas") serán irreducibles a la izquierda.

Y más adelante veremos que los conceptos de normalización conducen a una fe muy sencilla de declarar las DFs.

11.3 PRIMERA, SEGUNDA Y TERCERA FORMAS NORMALES

Advertencia: *Por razones de simplicidad, a lo largo de esta sección daremos por hecho cada varrel tiene exactamente una clave candidato, la cual además suponemos que es la á. primaria. Estas suposiciones se reflejan en nuestras definiciones, las cuales (repetimos) no muy rigoristas. El caso de una varrel que tenga más de una clave candidata lo expondremos la sección 11.5.*

Ahora estamos en posición de describir las tres formas normales originales de C(Primero, presentamos una definición preliminar (muy informal) de las varrels 3FN, para dar \ idea del punto al que queremos llegar. Después consideramos el proceso de reducir una vane cualquiera a una colección equivalente de varrels 3FN y damos sobre la marcha definiciones *m* precisas de las tres formas. Sin embargo, señalamos desde el principio que 1FN, 2FN y 3FN no son en sí mismas muy importantes salvo como escalones para llegar a FNBC (y más allá).

Aquí tenemos entonces nuestra definición preliminar de 3FN:

- **Tercera forma normal** *(definición muy informal):* una varrel está en 3FN si y sólo si los atributos que no son claves (si los hay) son
  a. Mutuamente independientes, y
  b. Dependientes irreduciblemente sobre la clave primaria.

Explicaremos (en general) los términos "atributo que no es clave" y "mutuamente independiente' como sigue:

- Un *atributo que no es clave* es cualquier atributo que no participa en la clave primaria (i la varrel respectiva.
- Dos o más atributos son *mutuamente independientes* si ninguno de ellos es dependiente funcionalmente de cualquier combinación de los otros. Tal independencia implica que cada uno de dichos atributos puede ser actualizado independientemente del resto.

A manera de ejemplo, la varrel de partes P está en 3FN de acuerdo con la definición **anl** rior. Los atributos PARTE, COLOR, PESO y CIUDAD son todos independientes entre sí (es posible, por ejemplo, cambiar el color de una parte sin tener que cambiar al mismo tiempo su peso) y todos son dependientes irreduciblemente de la clave primaria {P#}.

La definición anterior (informal) de 3FN, puede ser interpretada aún más informalmente como sigue:

- **Tercera forma normal** *(definición todavía más informal):* una varrel está en la tercera forma normal (3FN) si y solamente si cada tupia consiste, en todo momento, en un valor de clave primaria que identifica a alguna entidad, junto con un conjunto de cero o más valores de atributos mutuamente independientes que describen de alguna forma a esa entidad.

Una vez más, la varrel P se ajusta a la definición: cada tupia de P consiste en un valord clave primaria (un número de parte) que identifica a cierta parte en la realidad, junto con cuatro

valores adicionales (nombre de la parte, color, peso y ciudad), cada uno de los cuales sirve para describir la parte y es independiente de todos los demás.

Veamos ahora el procedimiento de normalización. Comenzamos con una definición de la primera forma normal.

- **Primera forma normal:** una varrel está en ÍFN si y sólo si, en cada valor válido de esa varrel, toda tupia contiene exactamente un valor para cada atributo.

Esta definición establece simplemente que todas las varrels están siempre en ÍFN, lo que por supuesto es correcto. Sin embargo, una varrel que *solamente* está en la primera forma normal (es decir, una varrel ÍFN que no está a la vez en 2FN y por lo tanto tampoco en 3FN) tiene una estructura que es indeseable por diversas razones. Para ilustrar la idea, supongamos que la información relativa a proveedores y envíos, más que estar dividida en dos varrels V y VP, se engloba en una sola varrel como sigue:

```
PRIMERA { V#,  STATUS,  CIUDAD,  P#,  CANT  }
        PRIMARY KEY { V#,  P#  }
```

Esta varrel es una versión ampliada de VCP de la sección 11.1. Los atributos tienen el mismo significado, con excepción de que para efectos del ejemplo, presentamos una restricción adicional:

```
CIUDAD -» STATUS
```

(STATUS es dependiente funcionalmente de CIUDAD. El significado de esta restricción es que el status de un proveedor se determina mediante la ubicación de ese proveedor; por ejemplo, todos los proveedores de Londres *deben* tener un status de 20.) Además, ignoramos PROVEEDOR para simplificar. La clave primaria de PRIMERA es la combinación {V#,P#}; mostramos el diagrama DF más adelante en la figura 11.5.

Observe que este diagrama DF es (informalmente) "más complejo" que el de una varrel 3FN. Como indicamos en la sección anterior, un diagrama 3FN tiene flechas que parten *sólo* de las claves candidatas, mientras que un diagrama que no es 3FN, como el de PRIMERA, tiene flechas que parten de claves candidatas *junto con ciertas flechas adicionales* (y son precisamente esas flechas adicionales las que ocasionan el problema). De hecho, la varrel PRIMERA viola ambas condiciones a. y b. de la definición 3FN anterior. Los atributos que no son clave no son todos mutuamente independientes —puesto que STATUS depende de CIUDAD (una flecha adicional)— y no son todos dependientes irreduciblemente de la clave primaria, debido a que STATUS y CIUDAD son cada uno dependientes sólo de V# (dos flechas adicionales más).

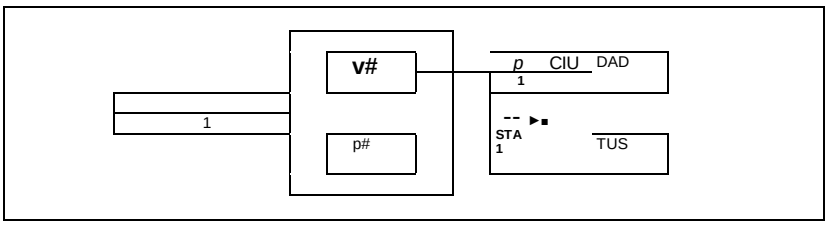


Figura 11.5 DFs para la varrel PRIMERA.

Como base para ilustrar algunas de las dificultades que surgen de esas flechas adiciónale la figura 11.6 presenta un valor de ejemplo para la varrel PRIMERA. Los valores de los atril: tos son básicamente los de costumbre, salvo que cambiamos el status del proveedor V3 de 30 10 para que fuera consistente con la nueva restricción de que CIUDAD determina a STATU! Las redundancias son obvias. Por ejemplo, toda tupia del proveedor V1 muestra la ciudad comí Londres; en forma similar, toda tupia de la ciudad Londres muestra el status 20.

PRIMERA

V#	STATUS	CIUDAD	P#	CANT
V1	20	Londres	P1	300
V1	20	Londres	P2	200
V1	20	Londres	P3	400
V1	20	Londres	P4	200
V1	20	Londres	P5	100
V1	20	Londres	P6	100
V2	10	París	P1	300
V2	10	París	P2	400
V3	10	París	P2	200
V4	20	Londres	P2	200
V4	20	Londres	P4	300
V4	20	Londres	P5	400

Figura **11.6** Valor de ejemplo de la varrel PRIMERA.

Las redundancias en la varrel PRIMERA conducen a una variedad de lo que por razone históricas se ha denominado regularmente como **anomalías de actualización;** es decir, dificultades con las operaciones de actualización INSERT, DELETE y UPDATE. Para ordenar m tras ideas, nos concentraremos primero en la redundancia proveedor-ciudad, que corresponde a la DF V# —> CIUDAD. Con cada una de las tres operaciones se presentan problemas:

- **INSERT:** No podemos insertar el hecho de que un proveedor en particular esté ubicado en una ciudad en particular hasta que ese proveedor suministre por lo menos una parte. De hecho, la figura 11.6 no muestra que el proveedor V5 está ubicado en Atenas. El motivo» que, hasta que V5 no suministre alguna parte, no tenemos un valor adecuado de la clave pri maria. (Al igual que en el capítulo 9, sección 9.4, también a lo largo de este capítulo damo por hecho —de manera bastante razonable— que los atributos de la clave primaria no tienen valores predeterminados. Vea el capítulo 18.)
- **DELETE:** Si eliminamos de PRIMERA la única tupia de un proveedor en particular, no sol eliminamos el envío que conecta a ese proveedor con una parte en particular sino también la información de que el proveedor está ubicado en una ciudad en particular. Por ejemplo, si eliminamos de PRIMERA la tupia con el valor V3 en V# y el valor P2 en P#, perdemos la información de que V3 está ubicado en París. (Los problemas de INSERT y DELETE re presentan en realidad dos caras de la misma moneda.)

  Nota: El verdadero problema aquí es que la varrel PRIMERA contiene demasiada información incluida; de ahí que cuando eliminamos una tupia, *eliminamos demasiado.* Para ser más específicos, la varrel PRIMERA contiene información con respecto a los envíos información alusiva a los proveedores, por lo que eliminar un envío hace que se elimine también la información del proveedor. Por supuesto, la solución a este problema consiste en "separar"; es decir, colocar la información de envíos en una varrel y la información d

proveedores en otra (y esto es exactamente lo que haremos en un momento más). Por lo tanto, otra manera informal de caracterizar el procedimiento de normalización es como un procedimiento de *separación:* colocar en varrels distintas, la información lógicamente distinta.

- **UPDATE:** En general, el valor de una determinada ciudad aparece varias veces en la varrel PRIMERA. Esta redundancia genera problemas de actualización. Por ejemplo, si el proveedor VI se cambia de Londres a Amsterdam, nos enfrentamos *ya sea* al problema de examinar PRIMERA para encontrar todas las tupias que conectan a VI con Londres (y cambiarlo), *o bien* a la posibilidad de producir un resultado inconsistente (como dar a la ciudad de VI el valor de Amsterdam en una tupia y de Londres en otra).

La solución a estos problemas, como ya sugerimos, es reemplazar la varrel PRIMERA por las dos varrels siguientes

SEGUNDA { V#, STATUS, CIUDAD }

VP { V#, P#, CANT }

La figura 11.7 muestra los diagramas de DFs para estas dos varrels; la figura 11.8 da valores de ejemplo. Observe que ahora se incluye la información del proveedor V5 (en la varrel SEGUNDA mas no en la varrel VP). De hecho, ahora la varrel VP es exactamente nuestra varrel de envíos ya conocida.

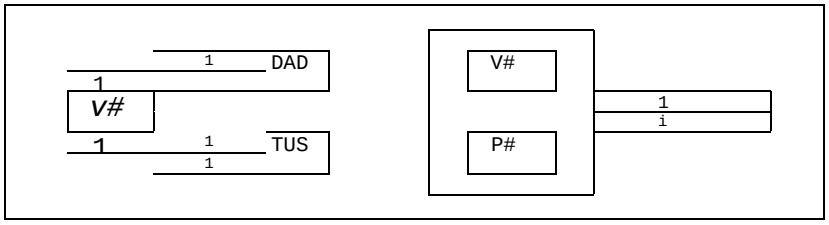


Figura **11.7** DFs para las varrels SEGUNDA y VP.

SEGUNDA

V#	STATUS	CIUDAD
V1	20	Londres
V2	10	París
V3	10	París
V4	20	Londres
V5	30	Atenas

VP

V#	P#	CANT
V1	P1	300
V1	P2	200
V1	P3	400
V1	P4	200
V1	P5	100
V1	P6	100
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200
V4	P4	300
V4	P5	400

Figura 11.8 Valores de ejemplo para las varrels SEGUNDA y VP.

Debe quedar claro que esta estructura modificada supera todos los problemas con op ciones de actualización antes esbozados:

- **INSERT:** Podemos insertar la información de que V5 está ubicado en Atenas, aunque a tualmente no suministre parte alguna, si simplemente insertamos la tupia correspondienl en SEGUNDA.
- **DELETE:** Podemos eliminar el envío que conecta a V3 y P2 si eliminamos la tupia cora pondiente de VP; no perdemos la información de que V3 está ubicado en París.
- **UPDATE:** En la estructura revisada, la ciudad de un determinado proveedor aparece uní sola vez —en lugar de muchas— debido a que existe precisamente una tupia para proveedor dado en la varrel SEGUNDA (la clave primaria de esa relación es {V#));eni palabras, se eliminó la redundancia V#-CIUDAD. Por lo tanto, podemos cambiar la ciudi para VI de Londres a Amsterdam modificándola de una vez por todas en la tupia relévame de SEGUNDA.

Al comparar las figuras 11.7 y 11.5, vemos que el efecto de la descomposición de PR MERA en SEGUNDA y VP, ha sido la eliminación de las dependencias que no eran irreducible (y es esta eliminación la que resolvió las dificultades). De manera intuitiva, podemos **decirqut** en la varrel PRIMERA el atributo CIUDAD no describía la entidad identificada por la clave primaria; es decir, un envío. En su lugar, describía al *proveedor* involucrado en ese envío (y (manera similar, por supuesto, para el atributo STATUS). Lo que ocasionó el problema fui primer lugar la combinación de las dos clases de información en una sola varrel.

Ahora damos una definición de la segunda forma normal:*

- **Segunda forma normal** *(definición que toma sólo una clave candidata, la cual supona que es la clave primaria):* Una varrel está en 2FN si y sólo si está en ÍFN y todo atrib que no sea clave es dependiente irreduciblemente de la clave primaria.

Las varrels SEGUNDA y VP están en 2FN (las claves primarias son (V#) y lacombinac {V#,P#}, respectivamente). La varrel PRIMERA no está en 2FN. Una varrel que está en la primera forma normal y no en la segunda siempre puede reducirse a una colección equivalen de varrels 2FN. El proceso de reducción consiste en reemplazar la varrel ÍFN por proyeccione convenientes; la colección de proyecciones así obtenida es equivalente a la varrel original (ei sentido de que siempre es posible recuperar esa varrel original juntando de nuevo dichas proy ciones). En nuestro ejemplo, SEGUNDA y VP son proyecciones de PRIMERA/ y PRIMERJ es la junta de SEGUNDA y VP sobre V#.

---

*Estrictamente hablando, 2FN sólo puede ser definida *con respecto a un conjunto especificado di dtp dencias,* pero es común ignorar este punto en los contextos informales. Se aplican observaciones simik a todas las formas normales (salvo la primera, por supuesto).

[i]Con excepción del hecho de que SEGUNDA puede incluir tupias (como la tupia del proveedor V5 de figura 11.8) que no tienen contraparte en PRIMERA. En otras palabras, la nueva estructura puede re] sentar información que podría no estar representada en la anterior. En este sentido, la nueva estructurapu« ser considerada como una representación ligeramente más fiel de la realidad.

Para resumir, el primer paso del procedimiento de normalización es tomar proyecciones para eliminar las dependencias funcionales "no irreducibles". Por lo tanto, dada la varrel *R* como sigue

```
{ A, B, C, D y
PRIMARY KEY { A, B }
/* damos por hecho que A      D es válida */
```

la disciplina de la normalización recomienda reemplazar *R* por sus dos proyecciones *R1* y *R2*, como sigue:

```
R1 { A, D }
   PRIMARY KEY { A }

R2 { A, B, C }
   PRIMARY KEY { A, B }
   FOREIGN KEY { A } REFERENCES R1
```

R puede ser recuperada tomando la junta "de clave externa a clave primaria coincidente" de *R2 yRl*.

Sin embargo, para regresar al ejemplo: la estructura SEGUNDA-VP todavía ocasiona problemas. La varrel VP es satisfactoria; de hecho, la varrel VP está ahora en 3FN y la ignoraremos por lo que resta de esta sección. Por otra parte, la varrel SEGUNDA sufre por la falta de independencia mutua entre sus atributos que no son clave. El diagrama DF de SEGUNDA es aún "más complejo" que un diagrama 3FN. Para ser específicos, la dependencia de STATUS sobre V#, aunque *es* funcional —y de hecho irreducible—, es **transitiva** (a través de CIUDAD). Cada valor V# determina un valor de CIUDAD y a su vez, cada valor de CIUDAD determina el valor de STATUS. De manera más general, siempre que las dos DFs *A* —> *B* y *B* —> *C* son válidas, entonces es una consecuencia lógica que la DF transitiva A —> *C* también es válida (como explicamos en el capítulo 10). Y las dependencias transitivas conducen una vez más a anomalías de actualización. (Ahora podemos concentrarnos en la redundancia ciudad-status, correspondiente a la DF CIUDAD -> STATUS.)

- **INSERT:** No podemos insertar el hecho de que una ciudad en particular tiene un status en particular; por ejemplo, no podemos declarar que cualquier proveedor en Roma deba tener un status de 50, hasta que en realidad tengamos un proveedor ubicado en esa ciudad.
- **DELETE:** Si eliminamos de SEGUNDA la única tupia de una ciudad en particular, no sólo eliminamos la información del proveedor respectivo, sino también la información de que esa ciudad tiene ese status en particular. Por ejemplo, si eliminamos de SEGUNDA la tupia de V5, perdemos la información de que el status de Atenas es 30. (Una vez más, los proble mas de INSERT y DELETE representan en realidad dos caras de la misma moneda.)

  Nota: Por supuesto, el problema es una vez más el exceso de información. La varrel SEGUNDA contiene información relativa a proveedores *e* información relativa a ciudades. Y por supuesto (de nuevo) la solución es "separar"; es decir, colocar la información de proveedores en una varrel y la información de ciudades en otra.
- UPDATE: En general, el status de una determinada ciudad aparece muchas veces en SE GUNDA (la varrel todavía contiene cierta redundancia). Por lo tanto, si necesitamos cam biar el status para Londres de 20 a 30, nos enfrentamos *ya sea* al problema de examinar SEGUNDA para encontrar todas las tupias de Londres (y modificarlas), *o bien* a la posi bilidad de producir un resultado inconsistente (al status para Londres podría dársele 20 en una tupia y 30 en otra).

De nuevo, la solución a estos problemas es reemplazar la varrel original (SEGUNDA) pe dos proyecciones; es decir, las proyecciones

```
VC { V#, CIUDAD }
```

```
CS { CIUDAD, STATUS }
```

La figura 11.9 muestra los diagramas de DFs para estas dos varrels; la figura 11.10 da val de ejemplo. Observe que en la varrel CS incluimos la información de status para Roma. Una vez más, la reducción es reversible, ya que SEGUNDA es la junta de VC y CS sobre CIUDAD.

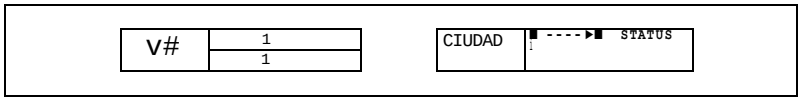


Figura 11.9 DFs para las varrels VC y CS.

VC

V#	CIUDAD
V1	Londres
V2	París
V3	París
V4	Londres
V5	Atenas

CS

CIUDAD	STATUS
Atenas	30
Londres	20
París	10
Roma	50

Figura 11.10 Valores de ejemplo para las varrels VC y CS.

De nuevo, debe quedar claro que esta estructura modificada supera todos los problemas coi operaciones de actualización bosquejados anteriormente. Dejamos como ejercicio la conside- ración detallada de dichos problemas. Al comparar las figuras 11.9 y 11.7, vemos que el efecto de la descomposición adicional es eliminar las dependencias transitivas de STATUS sobre VI (y es de nuevo esta eliminación la que resolvió las dificultades). Podemos decir, de manera i tuitiva, que en la varrel SEGUNDA el atributo STATUS no describía la entidad identificada por la clave primaria (es decir, un proveedor); describía en su lugar la ciudad en la que el proveedo estaba ubicado. Una vez más, la combinación de estos dos tipos de información en la misn varrel fue la que ocasionó los problemas.

Ahora damos una definición de la tercera forma normal:

- **Tercera forma normal** *(definición que toma sólo una clave candidato, la cual ademi tomamos como la clave primaria):* Una varrel está en 3FN si y sólo si está en 2FN y ti los atributos que no son clave son dependientes en forma no transitiva de la clave primaria. *Nota:* "Dependencias no transitivas" implica dependencias no *mutuas,* en el sentido del tér mino que explicamos casi al principio de esta sección.

Las varrels VC y CS están en 3FN (las claves primarias son {V#} y (CIUDAD), respectivamente). La varrel SEGUNDA no está en 3FN. Una varrel que está en la segunda forma normal y no en la tercera, siempre puede reducirse a una colección equivalente de varrels 3FN. Ya indicamos que el proceso es reversible y por lo tanto, que no perdemos información en la reducción; sin embargo, la colección 3FN puede contener información —como el hecho de que el status de Roma es 50— que no podría ser representada en la varrel 2FN original.*

Para resumir, el segundo paso en el procedimiento de normalización consiste en tomar proyecciones para eliminar las dependencias transitivas. En otras palabras, dada la varrel *R* como sigue

```
R { A, B, C }
  PRIMARY KEY { A }
  /* damos por hecho que S      C es válida
                               */
```

la disciplina de la normalización recomienda reemplazar *R* por sus dos proyecciones *R1* y *R2*, como sigue:

```
R1 { B, C }
   PRIMARY KEY { S }

H2 { A, B }
   PRIMARY KEY { A }
   FOREIGN KEY { S } REFERENCES R1
```

R puede ser recuperada tomando la junta "de clave externa a clave primaria coincidente" de *R2 yRl*.

Concluimos esta sección subrayando la idea de que el nivel de normalización de una varrel determinada es un asunto de semántica, no sólo un asunto del valor que resulte tener esa varrel en un momento específico. En otras palabras, no es posible observar tan sólo el valor en un momento dado y decir si la varrel está en (digamos) 3FN; también es necesario conocer el significado de los datos —es decir, las dependencias— antes de poder emitir un juicio similar. Observe también que aun conociendo las dependencias, nunca es posible *demostrar* mediante el examen de un valor dado, que la varrel está en 3FN. Lo mejor que podemos hacer es mostrar que el valor en cuestión no viola ninguna de las dependencias. Si damos por hecho que no las viola, entonces el valor es *consistente con la hipótesis* de que la varrel está en 3FN; aunque desde luego, ese hecho no garantiza que la hipótesis sea válida.

11.4 CONSERVACIÓN DE LA DEPENDENCIA

Durante el proceso de reducción, es frecuente el caso de que una varrel dada pueda ser descompuesta en varias formas diferentes, sin pérdida. Considere una vez más la varrel SEGUNDA de la sección 11.3, con las DFs V# -> CIUDAD y CIUDAD -> STATUS (y por lo tanto, por transitividad, también con V# —>ST ATUS; consulte la figura 11.11 en la que mostramos la DF transitiva como una flecha punteada).

---

*Deducimos que así como la combinación SEGUNDA-VP fue una representación de la realidad ligeramente mejor que la varrel 1FN PRIMERA, la combinación VC-CS es una representación ligeramente mejor que la varrel 2FN SEGUNDA.

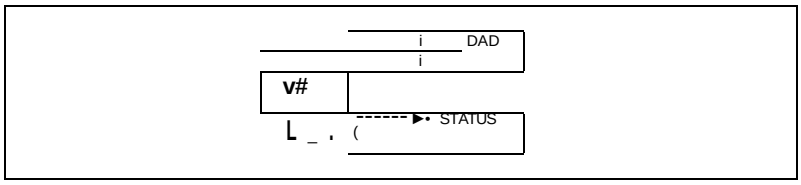


Figura 11.11 DFs para la varrel SEGUNDA.

En la sección 11.3 mostramos que las anomalías de actualización encontradas con SEGUNDA podían ser superadas si la reemplazábamos por su descomposición en las dos proyecciones 3FN

```
ve { v#, CIUDAD }
CS { CIUDAD, STATUS }
```

Nos referiremos a esta descomposición como "descomposición *A".* En contraste, aquítenemoi una descomposición alternativa ("descomposición *B"):*

```
VC { V#, CIUDAD }
VS { V#, STATUS }
```

(la proyección VC es la misma en ambos casos.) La descomposición *B* es también sin pérdida las dos proyecciones están una vez más en 3FN. Pero la descomposición *B* es menos satisfactoria que la A por diversas razones. Por ejemplo, aún no es posible insertar en *B* la información d que una ciudad en particular tiene un status determinado, a menos que se ubique un proveedor en esa ciudad.

Examinemos este ejemplo un poco más de cerca. Primero, observe que las proyecciones en la descomposición *A* corresponden a las flechas *continuas* de la figura 11.11, mientras que una de las proyecciones en la descomposición *B* corresponde a la flecha *punteada.* De hecho, en la descomposición *A,* las dos proyecciones son **independientes** entre sí, en el siguiente sentido: es pos ble hacer actualizaciones a cualquiera de ellas sin considerar a la otra.* Si suponemos que dichi actualización sólo es válida dentro del contexto de la proyección respectiva, lo cual sólo signific que no debe violar la restricción de unicidad de la clave primaria de esa proyección, entonces Id *junta de las dos proyecciones después de la actualización siempre será una SEGUNDA válida(ts* decir, no hay forma de que la junta viole las restricciones de DF sobre SEGUNDA). En contra en la descomposición *B,* es necesario vigilar las actualizaciones a cualquiera de las dos **proy** ciones para asegurar que no se viole la DF CIUDAD —> STATUS (si dos proveedores tienen la misma ciudad, entonces deben tener el mismo status; por ejemplo, considere lo que está implicado en la descomposición *B* al pasar el proveedor VI de Londres a París). En otras palabras, las dos proyecciones de la descomposición *B* no son independientes entre sí.

El problema básico es que en la descomposición *B,* la DF CIUDAD -> STATUS fue convertida, para usar la terminología del capítulo 8, en una *restricción de base de datos* que abare dos varrels (lo que implica, por cierto, que en muchos productos actuales se le tenga que ma tener mediante un código de procedimientos). En contraste, en la descomposición A. la DF

*Por supuesto, con excepción de la restricción referencial de VC a CS.

transitiva V# —» STATUS fue convertida en una restricción de base de datos y es esa restricción la que se hará cumplir automáticamente si se cumplen las dos restricciones *de varrel* V# —> CIUDAD y CIUDAD —> STATUS. Y por supuesto, es muy sencillo hacer cumplir estas dos últimas restricciones, lo cual implica (como es el caso) sólo hacer cumplir las restricciones correspondientes de unicidad de la clave primaria.

Por lo tanto, el concepto de proyecciones independientes proporciona un lineamiento para elegir una descomposición en particular cuando existe más de una posibilidad. De manera específica, es preferible una descomposición en la que las proyecciones sean independientes en el sentido que describimos arriba, en vez de una en la que no lo sean. Rissanen [11.6] muestra que las proyecciones *Rl* y *R2* de la varrel *R* son independientes en el sentido anterior si y solamente si:

- Toda DF en *R* es una consecuencia lógica de las DFs en *Rl* y *R2,* y
- Los atributos comunes de *Rl* y *R2* forman una clave candidata para al menos una de las dos proyecciones.

Considere las descomposiciones *A y B* como las definimos anteriormente. En *A* las dos proyecciones son independientes, ya que su atributo en común, CIUDAD, constituye la clave primaria de CS y toda DF en SEGUNDA aparece ya sea en una de las dos proyecciones o bien es una consecuencia lógica de las que aparecen. En contraste, en *B* las dos proyecciones no son independientes, ya que la DF CIUDAD —> STATUS no puede deducirse a partir de las DFs de esas proyecciones (aunque es cierto que su atributo en común, V#, constituye una clave candidata para ambas). *Nota:* La tercera posibilidad, reemplazar SEGUNDA por sus dos proyecciones sobre {V#,STATUS} y {CIUDAD,STATUS}, no es una descomposición válida ya que no es sin pérdida. *Ejercicio:* Demuestre esta declaración.

Decimos que una varrel es **atómica** cuando no puede ser descompuesta en proyecciones independientes [11.6]. Sin embargo, observe con cuidado que el hecho de que una varrel determinada no sea atómica en este sentido, no necesariamente debe tomarse como que debe ser descompuesta en componentes atómicos. Por ejemplo, las varrels V y P de la base de datos de proveedores y partes no son atómicas, aunque parece que no tiene mucho caso descomponerlas más. En contraste, la varrel VP *es* atómica.

La idea de que el procedimiento de normalización debe descomponer las varrels en proyecciones que sean independientes —en el sentido señalado por Rissanen— ha llegado a conocerse como **conservación de la dependencia.** Cerramos esta sección explicando con más precisión este concepto.

1. Suponga que se nos da una cierta varrel *R,* la cual después de aplicarle todos los pasos del procedimiento de normalización, reemplazamos por un conjunto de varrels *Rl, R2, ..., Rn* (por supuesto, todas ellas proyecciones de *R).*
2. Sea *S* el conjunto de DFs dadas para la varrel original *R,* y sean *SI, S2, ..., Sn* los conjuntos de DFs que se aplican a las varrels *Rl, R2, ..., Rn,* respectivamente.
3. Cada DF del conjunto *Si* se referirá solamente a los atributos de *Ri (i* = 1,2,..., *ri).* Entonces, resulta sencillo hacer cumplir las restricciones (DFs) en cualquier conjunto 5/ dado. Pero lo que necesitamos es hacer cumplir las restricciones en el conjunto original *S.* Por lo tanto, nos gustaría que la descomposición en *Rl, R2, ..., Rn* fuese tal, que hacer cumplir las restricciones en *SI, S2, ..., Sn* de manera individual, fuese en conjunto equivalente a hacer cumplir las restricciones en el conjunto original *S;* en otras palabras, quisiéramos que la descomposición *conservara la dependencia.*

4. Sea 5"la unión de *SI, S2, ..., Sn.* En general, observe que *no* se da el caso que *S'=S;* si embargo, para que la descomposición conserve la dependencia, es suficiente que los de *S* y *S'*sean iguales (si necesita refrescar su memoria con respecto a la noción de cierre< un conjunto de DFs, consulte la sección 10.4).
5. En general, no hay una forma eficiente de calcular el cierre S^+ de un conjunto de DFs;(manera que en realidad no es factible calcular los dos cierres y comparar su igualdad. Sin embargo, existe una forma eficiente de probar si una descomposición dada consérvala pendencia. Los detalles del algoritmo están fuera del alcance de este capítulo; para cono cerlos consulte, por ejemplo, el libro de Ullman [7.13].

Nota: La respuesta al ejercicio 11.3 al final del capítulo ofrece un algoritmo mediante cual una varrel arbitraria puede ser descompuesta sin pérdida (en una forma que conserva lac pendencia) en un conjunto de proyecciones 3FN.

11.5 FORMA NORMAL DE BOYCE/CODD

En esta sección dejamos la suposición (que utilizamos por razones de simplicidad) de que loda varrel tiene una sola clave candidata y consideraremos lo que sucede en el caso general. El hecho es que la definición original de Codd de la 3FN [10.4] no trataba satisfactoriamente el caso general. Para ser más precisos, no trataba adecuadamente el caso de una varrel que

1. tenía dos o más claves candidatas, tales que,
2. las claves candidatas estaban compuestas, y
3. se traslapaban (es decir, tenían al menos un atributo en común).

Por lo tanto, la definición original de 3FN fue reemplazada después por una definición raí sólida, debida a Boyce y Codd, la cual atendía también este caso [11.2]. Sin embargo, puesto que la nueva definición en realidad define una forma normal que es estrictamente más sólida que la antigua 3FN, es mejor presentar un nuevo nombre pare ella en lugar de seguirla llamando 3FN; de ahí el nombre *deforma normal de Boyce/Codd* (FNBC). * *Nota:* En la práctica las condiciones 1, 2 y 3 podrían no ser muy frecuentes. Para una varrel en donde no suceden, las 3FN y FNBC son equivalentes.

Con el fin de explicar la FNBC, primero le recordamos el término **determinante,** que introdujimos en el capítulo 10, para hacer referencia a la parte izquierda de una DF. También le recordamos el término **DF trivial,** que es una DF en la cual la parte izquierda es un superconjunto de la parte derecha. Ahora podemos definir la FNBC:

- **Forma norma) de Boyce/Codd:** Una varrel está en FNBC si y solamente si toda DFi trivial, irreducible a la izquierda, tiene una clave candidata como su determinante.

---

*Una definición de "tercera" forma normal, que de hecho era equivalente a la definición de la FNBC. fue introducida por primera vez por Heath en 1971 [11.4]; por lo tanto, hubiese sido un nombre más ap; "Forma normal de Heath".

O de manera menos formal:

■ **Forma normal de Boyce/Codd** *(definición informal):* Una varrel está en FNBC si **y** sólo si los únicos determinantes son claves candidatas.

En otras palabras, las únicas flechas en el diagrama DF son las que parten de las claves candidatas. Ya explicamos que siempre habrá flechas que parten de las claves candidatas; la definición de la FNBC dice que *no hay ninguna otra,* lo que significa que no se puede eliminar ninguna flecha mediante el proceso de normalización. *Nota:* La diferencia entre las dos definiciones FNBC es que asumimos tácitamente en el caso informal (a) que los determinantes "no son demasiado grandes" **y** (b) que todas las DFs son no triviales. Para simplificar, seguiremos haciendo estas suposiciones en el resto de este capítulo, excepto cuando indiquemos lo contrario.

Vale la pena señalar que la definición FNBC es conceptualmente más simple que la definición anterior de la 3FN, ya que no hace referencias explícitas a la primera **y** segunda formas normales como tales, ni al concepto de dependencia transitiva. Además, aunque (como ya indicamos) la FNBC es estrictamente más sólida que la 3FN, se sigue dando el caso que cualquier varrel dada puede descomponerse sin pérdida en una colección equivalente de varrels FNBC.

Antes de considerar algunos ejemplos que comprendan más de una clave candidata, convenzámonos de que las varrels PRIMERA y SEGUNDA que no estaban en 3FN, tampoco están en FNBC; y de que las varrels VP, VC **y** CS que estaban en 3FN, están también en FNBC. La varrel PRIMERA contiene tres determinantes; es decir {V#}, {CIUDAD} y {V#,P#}. De éstos, sólo {V#,P#} es una clave candidata, de modo que PRIMERA no está en FNBC. De manera similar, SEGUNDA tampoco está en FNBC, ya que el determinante (CIUDAD) no es una clave candidata. Por otra parte, las varrels VP, VC **y** CS, están todas en FNBC, ya que en cada caso la sola clave candidata es el único determinante en la varrel.

Consideremos ahora un ejemplo que involucra dos claves candidatas inconexas; es decir, que no se traslapan. Suponga que en la varrel usual de proveedores V {V#,PROVEEDOR,STATUS.CIUDAD), {V#} y {PROVEEDOR} son todas claves candidatas (es decir, durante todo el tiempo se da el caso de que todo proveedor tiene un número de proveedor único **y** también un nombre de proveedor único). Sin embargo, demos por hecho (como en cualquier otra parte de este libro) que los atributos STATUS **y** CIUDAD son independientes mutuamente; es decir, ya no es válida la DF CIUDAD -> STATUS que introdujimos sólo para los fines de la sección 11.3. Entonces, el diagrama DF es como el que muestra la figura 11.12.

La varrel V está en FNBC. Sin embargo, aunque el diagrama DF parece "más complejo" que un diagrama 3FN, se sigue dando el caso que los únicos determinantes son claves candidatas; es

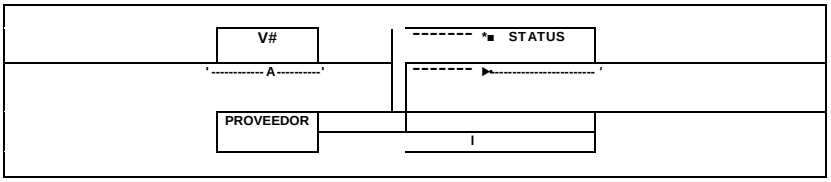


Figura 11.12 DFs para la varrel V si (PROVEEDOR) es una clave candidata (y ya no es válida CIUDAD -» STATUS).

decir, las únicas flechas son las que parten de claves candidatas. Así que el mensaje d< primer ejemplo es simplemente que no es necesariamente malo tener más de una clave i data. (Por supuesto, es necesario especificar ambas claves candidatas en la definición de la de datos, a fin de que el DBMS pueda hacer cumplir las restricciones requeridas de unicidad.

Ahora presentamos algunos ejemplos en los cuales las claves candidatas se traslapan, claves candidatas se traslapan si cada una involucra dos o más atributos y tienen por lo m un atributo en común. *Nota:* De acuerdo con nuestras explicaciones sobre este tema (vea el c¡ tulo 8), en ninguno de los ejemplos que siguen pretendemos elegir a una de las claves candidatas como la clave primaria. Por lo tanto, en esta sección tampoco marcaremos con doble subrayado ninguna de las columnas de nuestras figuras.

Para nuestro primer ejemplo, suponemos que los nombres de proveedor son únicos y consideramos la varrel

```
VVP { V#, PROVEEDOR, P#, CANT }
```

Las claves candidatas son {V#,P#} y {PRO VEEDOR,P#}. ¿Esta varrel está en FNBC? La respuesta es no, ya que contiene dos determinantes, V# y PROVEEDOR, que no son claves candidatas para la varrel ({V#} y {PROVEEDOR} son determinantes debido a que cadaur determina a la otra). Un valor de ejemplo para esta varrel lo da la figura 11.13.

VVP

V#	PROVEEDOR	P#	CANT
V1	Smith	P1	300
V1	Smith	P2	200
V1	Smith	P3	400
V1	Smith	P4	200

Figura 11.13 Valor de ejemplo (parcial) de la varrel WP.

Como muestra la figura, la varrel VVP involucra la misma clase de redundancias que las varrels PRIMERA y SEGUNDA de la sección 11.3 (y que la varrel VCP de la sección 11.1); de ahí que esté sujeta al mismo tipo de anomalías de actualización. Por ejemplo, cambiar el nombre del proveedor V1 de Smith a Robinson conduce una vez más a problemas de búsqueda o bien a resultados posiblemente inconsistentes. Sin embargo, de acuerdo a la definición antigua, VVP *está* en 3FN, ya que esa definición no requería que un atributo fuese dependiente —de manera irreducible— de cada clave candidata si éste era un componente de alguna clave candidata de la varrel y por lo tanto era ignorado el hecho de que PROVEEDOR no es dependiente irreduciblemente de {V#,P#}. *Nota:* Aquí, por "3FN" nos referimos a 3FN como se definió originalmente en la referencia [10.4], no a la forma simplificada que definimos en la sección 11.3.

Por supuesto, la solución a los problemas de VVP es dividir la varrel en dos proyecciones, en este caso las proyecciones

```
VV { V#, PROVEEDOR }
VP { V#, P#, CANT }
```

o de manera alternativa, las proyecciones

```
VV { V#, PROVEEDOR }
VP { PROVEEDOR, P#, CANT }
```

(en este ejemplo hay dos descomposiciones igualmente válidas). Todas estas proyecciones están en FNBC.

En este punto, probablemente nos detengamos un momento para reflexionar sobre lo que "en realidad" está sucediendo aquí. El diseño original, que consta únicamente de la varrel VVP, está *claramente* mal; sus problemas son obvios y no es probable que alguna vez fuera implementado por algún diseñador de bases de datos competente (aun cuando no se le hubiesen expuesto en lo absoluto las ideas de la FNBC y las demás). El sentido común diría que es mejor el diseño VV-VP. Pero ¿qué queremos decir con "sentido común"? ¿Cuáles son los *principios* que el diseñador aplica cuando elige el diseño VV-VP en lugar del diseño VVP?

Por supuesto, la respuesta es que son exactamente los principios de la dependencia funcional y la forma normal de Boyce/Codd. En otras palabras, esos conceptos (DF, FNBC y las demás ideas formales que explicamos en este capítulo y el siguiente) no son otra cosa que el *sentido común formalizado.* Toda la idea de la teoría subyacente a esta área consiste en tratar de identificar dichos principios de sentido común y formalizarlos (lo que desde luego no es algo fácil de conseguir). Pero si puede lograrse, entonces podemos *mecanizar* esos principios. En otras palabras, podemos escribir un programa y hacer que la máquina haga el trabajo. Por lo regular, los críticos de la normalización pasan por alto esta idea; ellos afirman (con bastante razón) que todas las ideas son en esencia mero sentido común, pero generalmente no se dan cuenta de que es un logro importante declarar en una manera precisa y formal lo que significa "sentido común".

Para retomar el hilo principal de nuestra explicación: como un segundo ejemplo del traslape de claves candidatas —debemos advertirle que algunas personas podrían considerar este ejemplo como patológico— consideramos una varrel EMP con los atributos E, M y P que representan estudiante, materia y profesor, respectivamente. El significado de una tupia EMP *{E:e,M:m,P:p}* es que al estudiante *e* se le enseña la materia *m* con el profesor *p.* Se aplican las siguientes restricciones:

- Para cada materia, a cada estudiante de esa materia le enseña un solo profesor.
- Cada profesor enseña sólo una materia (pero cada materia la enseñan varios profesores).

La figura 11.14 presenta un valor de ejemplo de EMP.

EMP

E	M	p
Smith	Matemáticas	Prof. White
Smith	Fisica	Prof. Green
Jones	Matemáticas	Prof. White
Jones	Fisica	Prof. Brown

Figura 11.14 Valor de ejemplo de la varrel EMP.

¿Cuáles son las DFs para la varrel EMP? De la primera restricción, tenemos la DF (E.M) -> P. De la segunda restricción, tenemos la DF P -> M. Por último, el hecho de que cada m la enseñen varios profesores nos indica que *no* es válida la DF M —> P. De modo que el diagra DF es como lo muestra la figura 11.15.

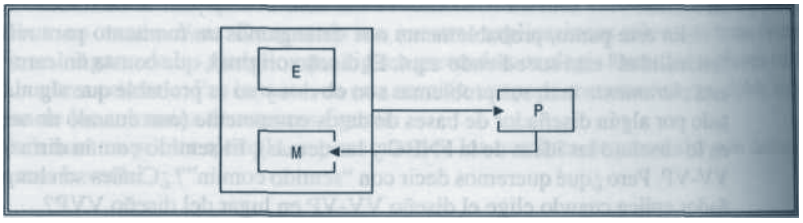


Figura 11.15 DFs para la varrel EMP.

De nuevo tenemos dos claves candidatas que se traslapan, es decir {E.IV1) y (E,P). L'navez más, la varrel está en 3FN y no en FNBC, y una vez más la varrel sufre por ciertas anómala actualización; por ejemplo, si queremos eliminar la información de que Jones está estudiando física, no podemos hacerlo sin perder al mismo tiempo la información de que el Profesor Brown enseña física. Estas dificultades son generadas por el hecho de que el atributo P es un determi nante pero no una clave candidata. De nuevo podemos superar los problemas sustituyéndola varrel original por dos proyecciones FNBC, en este caso las proyecciones

```
EP { E, P }
PM { P, M }
```

Dejamos como ejercicio mostrar los valores de estas dos varrels correspondientes a I datos de la figura 11.14, dibujar un diagrama DF correspondiente, demostrar que las dos proyecciones en realidad están en FNBC (¿cuáles son las claves candidatas?) y comprobar que la descomposición sí evita las anomalías.

Sin embargo, existe otro problema. El hecho es que aunque la descomposición en EPyPM sí evita ciertas anomalías, por desgracia ¡ introduce otras! El problema es que las dos proyecciones no son *independientes* en el sentido de Rissanen (vea la sección 11.4). Para ser específicos. laDF

```
{ E, M } -* P
```

no puede deducirse de la DF

(que es la única DF representada en las dos proyecciones). Como resultado, las dos proyecciones no pueden actualizarse de manera independiente. Por ejemplo, un intento por insertar una tupia para Smith y para el Prof. Brown en la varrel EP debe ser rechazado, ya que el Prof. Brown enseña física y a Smith ya le está enseñando física el Prof. Green; sin embargo, el sistema no puede detectar este hecho sin examinar la varrel PM. Por desgracia, nos vemos obligados a concluir

que los dos objetivos de (a) descomponer una varrel en componentes *FNBC* y (b) descomponerla en componentes *independientes,* pueden estar ocasionalmente en conflicto; es decir, no siempre es posible satisfacer al mismo tiempo ambos objetivos.

Nota: De hecho, la varrel EMP es *atómica* (vea la sección 11.4), incluso aunque no esté en FNBC. Por lo tanto, observe que el hecho de que una varrel atómica no pueda descomponerse en componentes independientes no significa que no pueda descomponerse en absoluto (donde por "descomponerse" entendemos —desde luego— descomponerse sin pérdida). Por lo tanto, la "atomicidad" no es en general un término muy bueno, ya que no es necesario ni suficiente para el buen diseño de bases de datos.

Nuestro tercero y último ejemplo sobre claves candidatas que se traslapan, se refiere a la varrel EXAMEN con los atributos E (estudiante), M (materia) y L (lugar). El significado de una tupia EXAMEN {E:e,M:w,L:/} es que el estudiante *e* fue examinado en la materia *m* y obtuvo el lugar / en la lista de la clase. Para los fines del ejemplo damos por hecho que la siguiente restricción es válida:

- No hay empates; es decir, dos estudiantes no pueden obtener el mismo lugar en la misma materia.

Las DFs son entonces como muestra la figura 11.16.

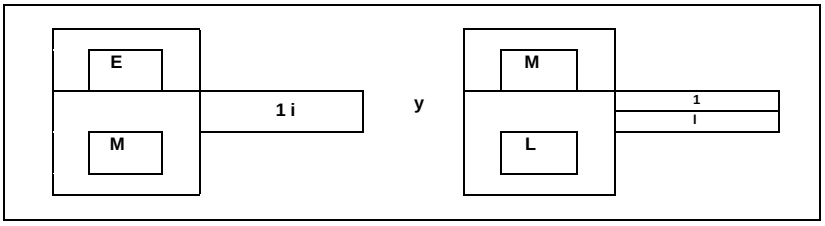


Figura 11.16 DFs para la varrel EXAMEN.

Una vez más tenemos dos claves candidatas que se traslapan (es decir {E,M} y {M,L}), ya que (a) si se nos da un estudiante y una materia entonces hay exactamente un lugar correspondiente y de igual forma, (b) si se nos da una materia y un lugar, hay exactamente un estudiante correspondiente. Sin embargo, la varrel está en FNBC, ya que dichas claves candidatas son los únicos determinantes y con esta varrel no ocurren anomalías de actualización como las que explicamos anteriormente en este capítulo. *(Ejercicio:* Compruebe esta afirmación.) Por lo tanto, traslapar claves candidatas *no necesariamente* conduce a problemas del tipo que hemos venido exponiendo.

En conclusión, vemos que el concepto de FNBC elimina ciertos casos de problemas adicionales que podrían ocurrir bajo la antigua definición de 3FN. Además, la FNBC es conceptualmente más sencilla que la 3FN, en el sentido de que no hace una referencia patente a los conceptos de 1FN, 2FN, clave primaria o dependencia transitiva. Lo que es más, la referencia que hace a las claves candidatas podría ser sustituida por una referencia a la noción más fundamental de dependencia funcional (de hecho, la definición dada en la referencia [11.2] hace este reemplazo).

Por otra parte, los conceptos de clave primaria y dependencia transitiva (entre otros) si en la práctica, ya que ofrecen una idea del proceso paso por paso real que el diseñador d seguir para poder reducir una varrel arbitraria a una colección equivalente de varreis FNBC. Finalmente comentamos que la respuesta al ejercicio 11.3 que se encuentra al final del ca tulo, incluye un algoritmo con el cual una varrel arbitraria puede ser descompuesta sin] en un conjunto de proyecciones FNBC.

11.6 UNA OBSERVACIÓN SOBRE LOS ATRIBUTOS CON VALOR DE RELACIÓN

En el capítulo 5 vimos que una relación puede incluir un atributo cuyos valores son a su vez relaciones (un ejemplo de ello lo muestra la figura 11.17). Por supuesto, como resultado las vaml también pueden tener atributos con valor de relación. Sin embargo, desde el punto de vista d diseño de bases de datos, existe una contraindicación para dichas varreis, ya que tienden a s *asimétricas** (¡por no mencionar el hecho de que sus predicados tienden a ser más bien compl cados!) y esa asimetría puede conducir a diversos problemas prácticos. Por ejemplo, en el cas de la figura 11.17 los proveedores y las partes son tratados de manera asimétrica. Como cons cuencia, las siguientes consultas (simétricas)

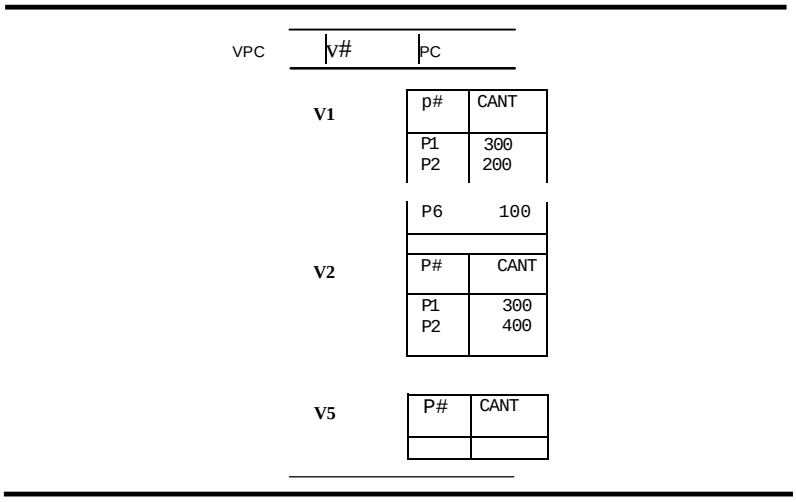


Figura 11.17 Una relación con un atributo con valor de relación.

*De hecho, históricamente dichas varreis ni siquiera eran válidas; se decía que eran *no normalizadas,* loque significaba que ni siquiera se les consideraba como parte de 1FN [10.4]. Vea el capítulo 5.

1. Obtener V# para los proveedores que suministran la parte Pl

2. Obtener P# para las partes que suministra el proveedor VI

tienen formulaciones muy diferentes:

```
1. ( VPC WHERE P# ( 'P1 ' ) IN PC { P# } ) { V# }

2. ( ( VPC WHERE V# = V# ( ' V1 ' ) ) { PC } ) { P# }
```

Aquí damos por hecho que VPC es una varrel cuyos valores son relaciones de la forma indicada por la figura 11.17.

Las cosas son aún peores para las operaciones de actualización. Por ejemplo, considere las dos actualizaciones siguientes:

1. Crear un nuevo envío para el proveedor V6, parte P5 y cantidad 500

2. Crear un nuevo envío para el proveedor V2, parte P5 y cantidad 500

Con nuestra varrel de envíos usual VP, no existe una diferencia cualitativa entre estas dos actualizaciones; ambas comprenden la inserción de una sola tupia en la varrel. En contraste, con la varrel VPC, las dos actualizaciones difieren de manera importante (por no mencionar el hecho de que *ambas* son mucho más complicadas que su contraparte VP):

```
1. INSERT INTO VPC RELATION
       { TUPLE { V# V# ( 'V6' ),
                 PC RELATION { TUPLE { P# P# ( 'P5' ),
                                       CANT CANT ( 500 )}}}};

2. UPDATE VPC WHERE V# = V# ( 'V2' )
     INSERT INTO PC RELATION { TUPLE { P# P# ( 'P5' ),
                                       CANT CANT ( 500 ) } } ;
```

Por lo tanto, preferimos las varrels (al menos las varrels base) sin atributos con valor de relación, debido a que tienen una estructura lógica más sencilla que conduce a simplificaciones correspondientes en las operaciones que necesitamos realizar en ellas. Sin embargo, entienda que esta posición debe ser vista solamente como un lineamiento, no como una ley inviolable. En la práctica, bien podría haber casos en que un atributo con valor de relación tuviera sentido, incluso para una varrel base. Por ejemplo, la figura 11.18 muestra (una parte de) un valor posible para una varrel *de catálogo* VRC que lista las varrels y sus claves candidatas. En esa varrel, el atributo CC tiene un valor de relación. Es además un componente de la única clave candidata de VRC. Una definición en **Tutorial D** para VRC podría entonces ser como la que sigue:

```
VAR VRC BASE RELATION
      { NOMBREVR NOMBRE, CC RELATION { NOMATRIBUTO NOMBRE } } KEY
        { NOMBREVR, CC } ;
```

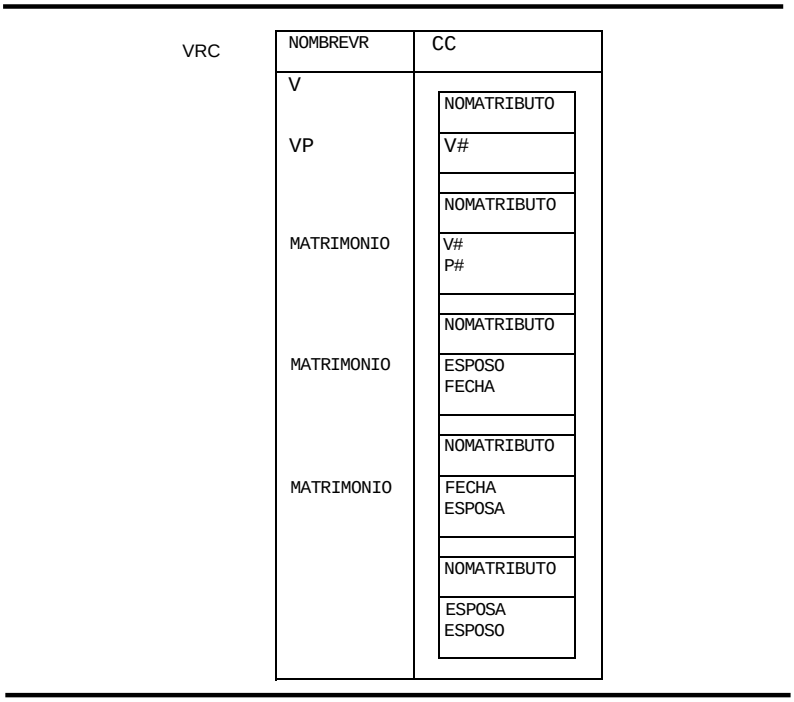


Figura 11.18 Valor de ejemplo para la varrel de catálogo VRC.

Nota: La respuesta al ejercicio 11.3 al final del capítulo muestra cómo eliminar atrito con valor de relación si dicha eliminación se considera necesaria (como generalmente sucede) Vea también la explicación del operador UNGROUP en el capítulo 6 (sección 6.8).

11.7 RESUMEN

Esto nos lleva al final del primero de nuestros dos capítulos sobre normalización adicional. Explicamos los conceptos de la **primera forma normal,** la **segunda,** la **tercera** y la de Boyce/Codd Las diversas formas normales (incluyendo la cuarta y la quinta, que serán explicadas en el siguiente capítulo) constituyen un *ordenamiento total,* en el sentido de que toda varrel en un determinadt nivel de normalización está también automáticamente en todos los niveles inferiores; aunque lo contrario no es cierto, ya que existen varrels en cada nivel que no están en ningún nivel superior Además, siempre es posible hacer una reducción a FNBC (y de hecho a 5FN); es decir, cualqi varrel dada siempre puede ser reemplazada por un conjunto equivalente de varrels en FNBC (o 5FN). La finalidad de dicha reducción es la de **evitar la redundancia** y por lo tanto, evitar ciertas **anomalías de actualización.**

*¡Y es posible! Observe que *no* es posible en el caso de VRC; al menos no directamente (es la introducción de alguna clase de atributo NOMBRECC, "nombre de clave candidata").

El proceso de reducción consiste en reemplazar la varrel dada por ciertas **proyecciones,** de tal manera que **juntar** de nuevo esas proyecciones nos regrese a la varrel original. En otras palabras, el proceso es **reversible** (o en forma equivalente, la descomposición es **sin pérdida).** Vimos también el papel crucial que las **dependencias funcionales** juegan en el proceso; de hecho, el **teorema de Heath** nos dice que si se satisface una cierta DF, entonces una cierta descomposición es sin pérdida. Este estado de las cosas puede ser visto como una confirmación adicional de la afirmación que hicimos en el capítulo 10 para el efecto de que las DFs "no son fundamentales, pero están cerca de serlo".

También explicamos el concepto de **proyecciones independientes** de Rissanen y sugerimos que es mejor (cuando exista la opción) descomponer en estas proyecciones que en otras que no sean independientes. Decimos que una descomposición en proyecciones independientes **conserva la dependencia.** Por desgracia, también vimos que los dos objetivos de (a) descomposición sin pérdida a FNBC y (b) conservación de la dependencia, pueden ocasionalmente presentar conflictos entre sí.

Concluimos este capítulo con un par de definiciones muy elegantes (y totalmente precisas) debidas a Zaniolo [11.7], de los conceptos de 3FN y FNBC. Primero, 3FN:

- **Tercera forma normal** *(definición de Zaniolo):* Sea *R* una varrel, sea *X* cualquier subconjunto de los atributos de *R* y sea *A* cualquier atributo individual de *R.* Entonces *R* está en 3FN si y sólo si, para toda DF *X* —> *A* en *R,* es cierto por lo menos alguno de los siguientes puntos:

  1. *X* contiene a A (así que la DF es trivial);
  2. Xes una superclave;
  3. *A* está contenida en una clave candidata de *R.*

La definición de la **forma normal de Boyce/Codd** se obtiene de la definición de 3FN al quitar simplemente la posibilidad número 3 (un hecho que muestra claramente que FNBC es estrictamente más sólida que 3FN). Por cierto, la posibilidad número 3 es precisamente la causa de lo "inadecuado" de la definición original de Codd de la 3FN [10.4], a la cual nos referimos en la introducción de este capítulo.

EJERCICIOS

11.1 Demuestre el teorema de Heath. ¿Es válido el inverso de este teorema?

11.2 A veces se afirma que toda varrel binaria está necesariamente en FNBC. ¿Es válida esta afirmación?

11.3 La figura 11.19 muestra la información a registrar en una base de datos con el personal de una compañía, representada como sería en un sistema *jerárquico* como el IMS (Information Management System) de IBM (vea el capítulo 1). La figura se lee como sigue:

- La compañía tiene un conjunto de departamentos.
- Cada departamento tiene un conjunto de empleados, un conjunto de proyectos y un conjunto de oficinas.
- Cada empleado tiene una historia laboral (conjunto de puestos que ha ocupado). Para cada uno de esos puestos, el empleado tiene también una historia salarial (conjunto de salarios recibidos mientras ocupó ese puesto).
- Cada oficina tiene un conjunto de teléfonos.

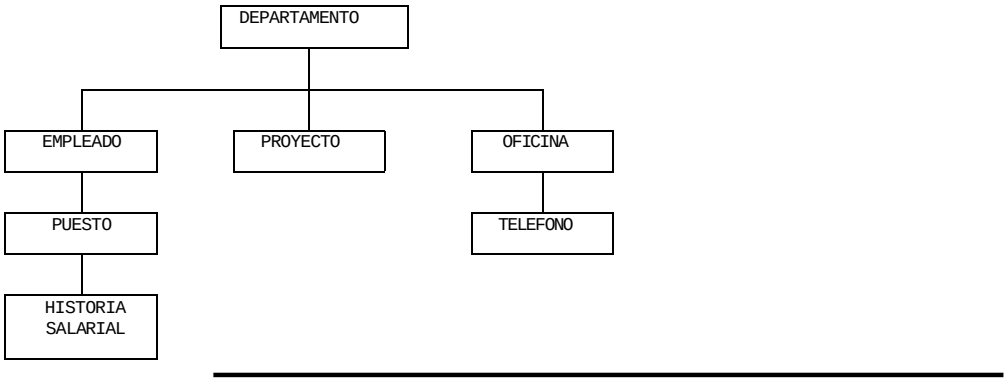


Figura 11.19 Una base de datos de una compañía (visión jerárquica).

La base de datos contendrá la siguiente información:

- Para cada departamento: número de departamento (único), presupuesto y número de empleado (único) del gerente del departamento;
- Para cada empleado: número de empleado (único), número de proyecto actual, número de ofi ciña y número telefónico; además, el nombre de cada puesto que ha ocupado el empleado mi la fecha y el salario para cada salario distinto recibido en ese puesto;
- Para cada proyecto: número de proyecto (único) y presupuesto;
- Para cada oficina: número de oficina (único), área del piso y números telefónicos (únicos) de todos los teléfonos de esa oficina.

Diseñe un conjunto adecuado de varrels para representar esta información. Declare cualquier suposición que haga con respecto a las dependencias funcionales.

11.4 En un sistema de registro de pedidos se emplea una base de datos conteniendo información sobre clientes, artículos y pedidos. Se incluirá la siguiente información:

- Para cada cliente:
  Número de cliente (único)
  Direcciones "Enviar a" (varias por cliente)
  Estado de cuenta
  Límite de crédito
  Descuento
- Para cada pedido:
  Información de encabezado: número de cliente
  dirección de envío
  fecha del pedido
  Líneas de detalle (varias por pedido): número de artículo
  cantidad ordenada
- Para cada artículo:
  Número de artículo (único)

Plantas manufactureras
Existencia en cada planta
Nivel de riesgo de almacenamiento para cada planta
Descripción del artículo

Además (por razones de procesamiento interno) a cada línea de detalle para cada pedido, se asocia un valor "cantidad pendiente"; a este valor se le asigna inicialmente la cantidad del artículo solicitado y (en forma progresiva) se reduce a cero conforme se hacen los envíos (parciales). De nuevo, diseñe una base de datos para esta información. Como en el ejercicio anterior, declare cualquier suposición que haga con respecto a las dependencias.

11.5 Suponga que en el ejercicio 11.4 sólo un número reducido de clientes (digamos el uno por ciento o menos) tiene en realidad más de una dirección de envío. (Ésta es una situación típica real, en la que es frecuente el caso que sólo unas cuantas excepciones —a menudo las importantes— no se apegan a cierto patrón general.) ¿Puede identificar algunos inconvenientes a su solución del ejercicio 11.4? ¿Puede imaginar algunas mejoras?

11.6 *(Versión modificada del ejercicio 10.13.)* La varrel HORARIO tiene los siguientes atributos:

D Día de la semana (1 a 5)
H Horas del día (1 a 8)
C Número de salón de clases
P Nombre del maestro
E Nombre del estudiante
L Nombre de la materia

La tupia *{D:d,H:h,C:c,P:p,E:e,L:l}* aparece en esta varrel si y sólo si en el momento *{D:d,H:h}* el estudiante *e* toma la materia / que es impartida por el maestro *p* en el salón *c*. Puede dar por hecho que las materias tienen una duración de un periodo (en horas) y que cada materia tiene un nombre que es único con respecto a las demás que se imparten en la semana. Reduzca HORARIO a una estructura más apropiada.

11.7 *(Versión modificada del Ejercicio 10.14.)* La varrel NDIR tiene los atributos NOMBRE (único), CALLE, CIUDAD, ESTADO y CP. Para cualquier código postal (CP) dado, sólo hay una ciudad y un estado. Además, para cualquier calle, ciudad y estado dados, sólo hay un código postal. ¿Está NDIR en FNBC? ¿En 3FN? ¿En 2FN? ¿Puede imaginar un mejor diseño?

REFERENCIAS Y BIBLIOGRAFÍA

Además de las siguientes, vea las referencias del capítulo 10; en especial los artículos originales de Codd sobre 1FN, 2FN y 3FN [10.4-10.5].

11.1 Philip A. Bernstein: "Synthesizing Third Normal Form Relations from Functional Dependencies", *ACM TODS 1,* No. 4 (diciembre, 1976).

En este capítulo explicamos algunas técnicas para descomponer varrels "grandes" en otras "más pequeñas" (es decir, unas de menor grado). En este artículo, Bernstein considera el problema inverso de usar varrels "pequeñas" para construir otras "más grandes" (es decir, de mayor grado). El problema en realidad no está caracterizado en esta forma dentro del artículo; más bien, es descrito como el problema de *sintetizar* varrels dado un conjunto de atributos y un conjunto de DFs correspondientes (con la restricción de que las varrels sintetizadas deben estar en 3FN). Sin embargo, puesto que los atributos y las DFs no tienen significado fuera del contexto de alguna varrel que los contenga, sería más preciso ver la construcción primitiva como una varrel binaria que contiene una DF, en lugar de como un par de atributos más una DF. *Nota:* De igual forma,

bien sería posible considerar al conjunto dado de atributos y DFs como si definieran a una va **universal** —vea, por ejemplo, la referencia [12.9]— que satisface un conjunto dado de DFs; e cuyo caso el proceso de "síntesis" puede ser percibido como un proceso para *descomponer tí* varrel universal en proyecciones 3FN. Aunque para fines de la presente exposición, nos q damos con la interpretación original de "síntesis".

Entonces, el proceso de síntesis consiste en construir varrels «-arias a partir de varrels b narias dado un conjunto de DFs que se aplica a dichas varrels y dado el objetivo de que todas la varrels construidas están en 3FN (cuando se hizo este trabajo, la FNBC aún no estaba definii El artículo también presenta algoritmos para realizar esta tarea.

Una objeción a este enfoque (reconocida por Bernstein) es que las manipulaciones realizadas por el algoritmo de síntesis son de naturaleza puramente sintáctica y no toman en cuenta la semántica. Por ejemplo, dadas las DFs

A -» **s** (para la varrel *R{A, B})*
B -> *c* (para la varrel *S{B,* C})
A -> *c* (para la varrel *T{A, C))*

la tercera podría o no ser redundante (es decir, estar implicada por la primera y la segunda), dependiendo del significado de *R, S* y *T.* Como un ejemplo en donde no está implicada, tome a A come número de empleado, a *B* como número de oficina y a *C* como número de departamento; tome ¡ como "oficina del empleado", a *S* como "departamento al que pertenece la oficina" y a 7"como' partamento del empleado"; y considere finalmente el caso de un empleado que trabaja en una oficina que pertenece a un departamento que no es el propio del empleado. El algoritmo de sínte simplemente da por hecho que, por ejemplo, los dos atributos *C* son uno mismo (de hecho, no reconoce en lo absoluto nombres de varrels); por lo tanto, depende de algún mecanismo externo (es decir, de la intervención humana) para evitar manipulaciones semánticamente inválidas. En el caso presente, sería responsabilidad de la persona que define las DFs originales usar nombres de atributo distintos (por decir algo) *Cl y C2* en las dos varrels *S* y *T.*

11.2 E. F. Codd: "Recent Investigations into Relational Data Base Systems". Proc. IFIP Congress. Estocolmo, Suecia (1974) y en otras partes.

Este artículo cubre en cierto modo una combinación de temas. Sin embargo, ofrece en particu lar "una definición mejorada de la tercera forma normal" (donde "tercera forma normal" se refiere de hecho a lo que ahora se conoce como forma normal de *Boyce/Codd).* Otros temas expuestos comprenden *vistas y actualización de vistas, sublenguajes de datos, intercambio de datos* e úi *vestigaciones necesarias* (todos a partir de 1974).

11.3 C. J. Date: "A Normalization Problem", en *Relational Database Writings 1991-1994.* Reading. Mass.: Addison-Wesley (1995).

Para citar el resumen, este artículo "examina un problema sencillo de normalización y loempl para hacer algunas observaciones sobre el tema del diseño de bases de datos y la declaraciónd algunas restricciones de integridad explícitas". El problema comprende una aplicación sencilla de una línea aérea y las siguientes DFs:

```
{ VUELO } -» DESTINO
{ VUELO } ->H0RA
{ DÍA, VUELO } -> PUERTA
{ DÍA, VUELO } ->PILOTO
{ DÍA, HORA, PUERTA } -> DESTINO
{ DÍA, HORA, PUERTA } -»VUELO
{ DÍA, HORA, PUERTA } -^PILOTO
{ DÍA, HORA, PILOTO} -> DESTINO
{ DÍA, HORA, PILOTO} ->VUELO
{ DÍA, HORA, PILOTO} ^PUERTA
```

Entre otras cosas, este ejemplo sirve para ilustrar la idea de que el diseño "correcto" de bases de datos rara vez puede decidirse con base exclusivamente en los principios de la normalización.

11.4 I. J. Heath: "Unacceptable File Operations in a Relational Database", Proc. 1971 ACM SIG-FIDET Workshop on Data Description, Access, and Control, San Diego, Calif, (noviembre, 1971).

Este artículo da una definición de "3FN" que de hecho fue la primera definición publicada de *FNBC.* También incluye una demostración de lo que en la sección 11.2 nos referimos como *teorema de Heath.* Observe que los tres pasos del proceso de normalización —tal como los explicamos en el cuerpo de este capítulo— son aplicaciones de ese teorema.

11.5 William Kent: "A Simple Guide to Five Normal Forms in Relational Database Theory", *CACM* 26, No. 2 (febrero, 1983).

El origen de la siguiente caracterización —intuitivamente atractiva— de "3FN" (para ser más precisos, de FNBC): **Cada atributo debe representar un hecho sobre la clave, toda la clave y nada más que la clave** (ligeramente parafraseada).

11.6 Jorma Rissanen: "Independent Components of Relations", *ACM TODS 2,* No. 4 (diciembre, 1977).

11.7 Carlo Zaniolo: "A New Normal Form for the Design of Relational Database Schemata", *ACM TODS* 7, No. 3 (septiembre, 1982).

El origen de las elegantes definiciones de 3FN y FNBC que mencionamos en la sección 11.7. La finalidad principal del artículo es definir otra forma normal, la *forma normal de clave elemental* (FNCE), que se ubica entre 3FN y FNBC y "capta las cualidades sobresalientes de ambas" mientras que evita los problemas que presentan (es decir, que 3FN es "demasiado complaciente" y FNBC es "propensa a la complejidad computacional"). El artículo también muestra que el algoritmo de Bernstein [11.1] genera de hecho varrels que están en FNCE y no sólo en 3FN.

RESPUESTAS A EJERCICIOS SELECCIONADOS

11.1 El teorema de Heath establece que si *R{A,B,C]* satisface laDFA —>B (donde *A,* fly C son con juntos de atributos), entonces *R* es igual a la junta de sus proyecciones *Rl* sobre *{A,B}* y *R2* sobre *{A,C}.* En la siguiente demostración del teorema, adoptamos nuestra forma abreviada usual para las tupias, al escribir, por ejemplo *(a,b,c)* para *{A:a,B:b,C:c}.*

Primero mostramos que no se pierde ninguna tupia de *R* tomando las proyecciones y después juntándolas de nuevo. Sea *(a,b,c)* e *R.* Entonces *(a,b)* e *Rl* y *(a,c)* e *R2,* y por lo tanto *(a,b,c)* e *Rl* JOIN *R2.* |

A continuación mostramos que toda tupia de la junta es en realidad una tupia de *R* (es decir, la junta no genera ninguna tupia "falsa"). Sea *(a,b,c)* e *Rl* JOIN *R2.* Con el fin de generar dicha tupia en la junta, debemos tener *(a,b)* e *Rl y (a,c)* e *R2.* De ahí que deba existir una tupia *(a,b',c)* e *R* para alguna *b'* a fin de generar la tupia *(a,c)* e *R2.* Por lo tanto, debemos tener *(a,b*^{r}*)* e *Rl.* Ahora bien, tenemos*(a,b)* e *R1* y *(a,b')* e *RT,* por lo tanto debemos tener *b = b',* ya que *A —> B.* De donde *(a,b,c) íR.* ■

El inverso del teorema de Heath establecería que si *R[A,B,C]* es igual a la junta de sus proyecciones sobre *{A,B)* y sobre *{A,C},* entonces *R* satisface la *DF A —> B.* Esta declaración es falsa. Por ejemplo, la figura 12.2 del siguiente capítulo muestra una relación que ciertamente es igual a la junta de sus proyecciones y sin embargo no satisface en absoluto ninguna DF (no trivial).

11.2 La afirmación es casi válida, pero no del todo. El siguiente contraejemplo patológico fue to mado de la referencia [5.5]. Considere la varrel EUA {PAIS,ESTADO}, que se interpreta como

"ESTADO es miembro de PAÍS"; donde PAÍS es Estados Unidos de América en cada tupia. Em ees, la DF

```
{ } -> PAÍS
```

es válida en esta varrel y sin embargo, el conjunto vacío (} no es una clave candidata. De modoqu EUA no está en FNBC (podemos descomponerla sin pérdida en sus dos proyecciones uñarías, aunque podría ser tema de debate el hecho de si debería normalizarse aún más de esta manera).

Por cierto, observe que en general es bastante posible tener una clave candidata que *sea* el conjunto vacío. Para una mayor explicación, vea la respuesta al ejercicio 8.7 del capítulo 8. **11.3** La figura 11.20 muestra las dependencias funcionales más importantes, tanto aquellas implíci tas por la descripción del ejercicio, como las correspondientes a aquellas suposiciones semánticas rs zonables (que enunciamos explícitamente más adelante). Pretendemos que los nombres de lo¡ atributos sean claros.

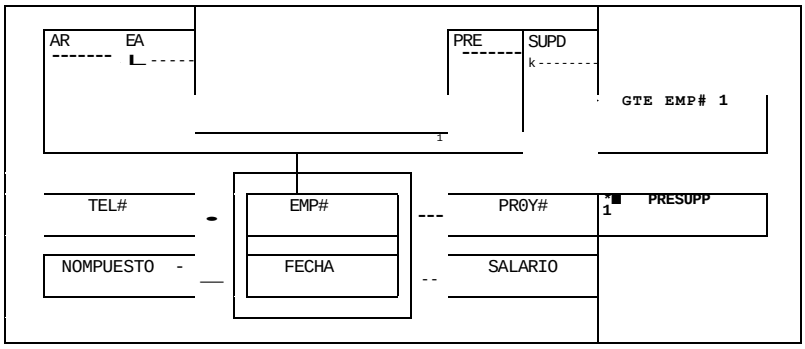


Figura 11.20 Diagrama de DFs del ejercicio 11.3.

Suposiciones semánticas:

- Ningún empleado es gerente de más de un departamento a la vez.
- Ningún empleado trabaja en más de un departamento a la vez.
- Ningún empleado trabaja en más de un proyecto a la vez.
- Ningún empleado tiene más de una oficina a la vez.
- Ningún empleado tiene más de un teléfono a la vez.
- Ningún empleado tiene más de un puesto a la vez.
- Ningún proyecto está asignado a más de un departamento a la vez.
- Ninguna oficina está asignada a más de un departamento a la vez.
- Los números de departamento, empleado, proyecto, oficina y teléfono son todos "globalment únicos.

Paso 0: Establecer una estructura inicial de varrel

Observe primero que la estructura jerárquica original puede ser considerada como una varrel DEPTOO con *atributos con valor de relación:*

```
DEPTO0 { DEPTO#, PRESUPD, GTE_EMP#, XEMP0, XPROY0, XOFICINA0 }
       KEY { DEPTO# } KEY { GTE_EMP# }
```

Los atributos DEPTO#, PRESUPD y GTE_EMP# se explican por sí mismos, pero los atributos XEMPO, XPROYO y XOFICINAO tienen valores de relación y requieren de una explicación adicional:

- El valor XPROYO dentro de una determinada tupia de DEPTOO es una relación con los atributos PROY# y PRESUPP.
- « En forma similar, el valor XOFICINAO dentro de una tupia dada de DEPTOO es una relación con los atributos OFICINA*, AREA y (digamos) XTELO; donde XTELO tiene a su vez un valor de relación. Las relaciones XTELO tienen sólo un atributo, TEL#.
- Por último, el valor XEMPO dentro de una tupia dada de DEPTOO es una relación con los atributos EMP#, PROY#, OFICINA*, TEL# y (digamos) XPUESTOO; donde XPUESTOO tiene a su vez un valor de relación. Las relaciones XPUESTOO tienen los atributos NOMPUESTO y (digamos) XHISTSALO; donde XHISTSALO tiene una vez más un valor de relación (las relaciones XHISTSALO tienen los atributos FECHA y SALARIO).

Por lo tanto, la jerarquía completa puede ser representada mediante la siguiente estructura anidada:

```
DEPTO0 { DEPTO», PRESUPD, GTE_EMP#,
         XEMP0 { EMP#, PROY#, 0FICINA#, TEL#,
                 XPUESTO0 { NOMPUESTO,
                            XHISTSAL0 { FECHA, SALARIO } } },
         XPROY0 { PROY#, PRESUPP }, XOFICINA0 { OFICINA*,
         AREA, XTEL0 { TEL* } } }
```

Nota: Aquí, en lugar de intentar mostrar las claves candidatas, usamos *cursivas* para indicar los atributos que por lo menos son "únicos dentro del padre" (de hecho, DEPTO#, EMP#, PROY#, OFICINA* y TEL# son, de acuerdo con las suposiciones que declaramos, *globalmente* únicos).

Paso 1: Eliminar los atributos con valor de relación

Ahora bien, para simplificar supongamos que deseamos que toda varrel tenga específicamente una clave *primaria*, es decir, siempre designaremos a una clave candidata como primaria por alguna razón (la razón no es importante en este momento). En el caso particular de DEPTOO, seleccionamos (DEPTO#) como la clave primaria (y así, {GTE_EMP#} se convierte en una clave alterna).

Ahora procederemos a deshacernos de todos los atributos con valor de relación de DEPTOO, ya que (como señalamos en la sección 11.6) dichos atributos son generalmente innecesarios:*

- » Para cada atributo con valor de relación de DEPTOO —es decir, los atributos XEMPO, XPROYO y XOFICINAO— formamos una nueva varrel con atributos que consisten en los atributos de las relaciones aplicables junto con la clave primaria de DEPTOO. La clave primaria de cada una de dichas varrels es la combinación del atributo que antes dio "la unicidad dentro del padre", junto con la clave primaria de DEPTOO. (Sin embargo, observe que muchas de esas "claves primarias" incluirán atributos que son redundantes para fines de identificación única y serán eliminados más adelante.) Quitar los atributos XEMPO, XPROYO y XOFICINAO de DEPTOO.

"Observamos que el procedimiento dado aquí para eliminar atributos con valor de relación equivale a ejecutar repetidamente el operador UNGROUP (vea el capítulo 6, sección 6.8) hasta obtener el resultado deseado. Por cierto, tal como describimos el procedimiento, garantiza además que también sean eliminadas todas las dependencias multivaluadas que no sean DFs; como consecuencia, las varrels con las que terminamos finalmente, están de hecho en 4FN, no solamente en FNBC (vea el capítulo 12).

- Si cualquier varrel *R* aún incluye cualquier atributo con valor de relación, realice sobre *R* una secuencia análoga de operaciones.

Obtenemos la siguiente colección de varrels, como indicamos, con todos los atributos con valor relación eliminados. Sin embargo, observe que mientras que las varrels resultantes están (supuesto) en 1FN, no necesariamente están en alguna forma normal superior.

```
DEPT01 { DEPT0#, PRESUPD, GTE_EMP# }
       PRIMARY KEY { DEPTO# } ALTERNATE
       KEY { GTE_EMP# }

EMP1 { DEPTO#, EMP#, PR0Y#, OFICINA*, TEL# }
      PRIMARY KEY { DEPTO#, EMP# }

PUESTO1 { DEPTO#, EMP#, NOMPUESTO }
        PRIMARY KEY { DEPTO#, EMP#, NOMPUESTO }

HISTSAL1 { DEPTO#, EMP#, NOMPUESTO, FECHA, SALARIO }
         PRIMARY KEY { DEPT0#, EMP#, NOMPUESTO, FECHA }

PR0Y1 { DEPTO#, PR0Y#, PRESUPP }
      PRIMARY KEY { DEPT0#, PR0Y# }

0FICINA1 { DEPTO#, OFICINA*, AREA }
         PRIMARY KEY { DEPT0#, OFICINA» }

TEL1 { DEPTO#, OFICINAL, TEL# }
     PRIMARY KEY { DEPTO#, OFICINA*, TEL# }
```

Paso 2: Reducir a 2FN

Ahora reducimos las varrels producidas en el paso I a una colección equivalente de varrels en 2FN, eliminando cualquier DF que no sea irreducible. Consideraremos las varrels una por una.

DEPTO1: Esta varrel ya está en 2FN.

EMP1: De hecho, observe primero que DEPTO# es redundante como un componente de la clave primaria de esta varrel. Podemos tomar sólo {EMP#} como la clave pri mana, en cuyo caso la varrel está en 2FN.

PUESTO 1: De nueva cuenta, no se requiere DEPTO# como componente de la clave primaria. Debido a que DEPTO# es dependiente funcionalmente de EMP#, tenemos un atributo que no es clave (DEPTO#) y que no es irreduciblemente dependiente de la clave primaria (la combinación {EMP#,NOMPUESTO)); de ahí que PUESTO 1 no esté en 2FN. Podemos reemplazarla por

```
PUEST02A { EMP#, NOMPUESTO }
         PRIMARY KEY { EMP#, NOMPUESTO }


PUEST02B { EMP#, DEPTO# }
         PRIMARY KEY { EMP# }
```

Sin embargo, PUESTO2A es una proyección de HISTSAL2 (vea adelante), y PUESTO2B es una proyección de EMP1 (renombrado adelante como EMP2), así que estas dos varrels pueden ser descartadas.

HISTSAL1: Al igual que con PUESTO 1, podemos proyectar a DEPTO# completamente hacia afuera. Lo que es más, NOMPUESTO no es necesario como un componente de la clave primaria; podemos tomar como clave primaria la combinación {EMP#,FECHA}, para obtener la varrel en 2FN

```
HISTSAL2 { EMP#, FECHA, NOMPUESTO, SALARIO }
         PRIMARY KEY { EMP#, FECHA }
```

rde
por

PROY1: Al igual que con EMP1, podemos considerar a DEPTO# como un atributo que no es clave; la varrel queda entonces en 2FN.

OFICINAL- Se aplican observaciones similares.

TELL Podemos proyectar a DEPTO# completamente hacia afuera, ya que la varrel (DEPTO#,OFICINA#) es una proyección de OFICINA1 (renombrada más adelante como 0FICINA2). Además, OFICINA* es dependiente funcionalmente de TEL#; así que podemos tomar solamente a {TEL#} como la clave primaria para obtener la varrel en 2FN

```
TEL2 { TEL#, OFICINA* >
     PRIMARY KEY { TEL# }
```

Observe que esta varrel no es necesariamente una proyección de EMP2 (podrían existir teléfonos u oficinas sin estar asignados a empleados), así que no podemos descartar esta varrel. Por lo tanto, nuestra colección de varrels 2FN es

```
DEPT02 { DEPT0#, PRESUPD, GTE^EMP* }
       PRIMARY KEY { DEPTO* }
       ALTERNATE KEY { GTE_EMP# }

EMP2 { EMP#, DEPTO#, PROY#, OFICINA*, TEL# }
     PRIMARY KEY { EMP# }

HISTSAL? / <siw, r¿im, m/rvEsm, SALARIO >
       PRIMARY KEY { EMP#, FECHA }

PR0Y2 { PROY#, DEPTO*, PRESUPP }
      PRIMARY KEY { PROY* }

0FICINA2 { 0FICINA#, DEPTO*, AREA }
         PRIMARY KEY { OFICINA* }

TEL2 { TEL#, OFICINAL }
     PRIMARY KEY { TEL* }
```

Paso 3: Reducir a 3FN

Ahora reducimos las varrels 2FN a un conjunto equivalente 3FN, eliminando las dependencias transitivas. La única varrel 2FN que todavía no está en 3FN es la varrel EMP2, en la cual OFICINA* y DEPTO# son dependientes transitivamente de la clave primaria {EMP#¡ (OFICINA* a través de TEL# y DEPTO* a través de PRO Y* y de OFICINA*; y por lo tanto, también a través de TEL#). Las varrels 3FN (proyecciones) correspondientes a EMP2 son

```
EMP3 { EMP#, PROY*, TEL# }
     PRIMARY KEY { EMP* }

X { TEL#, OFICINA* }
  PRIMARY KEY { TEL* }

Y { PR0Y#, DEPTO* }
  PRIMARY KEY { PROY* }

Z { OFICINA*, DEPTO* }
  PRIMARY KEY { OFICINA* }
```

Sin embargo, X es TEL2, Y es una proyección de PROY2 y Z es una proyección de OFICINA2. De ahí que nuestra colección de varrels 3FN sea simplemente

```
DEPT03 { DEPTO#, PRESUPO, GTE_EMP# }
       PRIMARY KEY { DEPT0# }
       ALTERNATE KEY { GTE_EMP# }

EMP3 { EMP#, PR0Y#, TEL# }
      PRIMARY KEY { EMP# }

HISTSAL3 { EMP#, FECHA, NOMPUESTO, SALARIO }
         PRIMARY KEY { EMP#, FECHA }

PR0Y3 { PROY#, DEPTO#, PRESUPP }
      PRIMARY KEY { PROY# }

0FICINA3 { OFICINA*, DEPTO#, AREA }
         PRIMARY KEY { OFICINA* }

TEL3 { TEL#, OFICINA)? }
     PRIMARY KEY { TEL# }
```

Por último, es fácil ver que cada una de estas varrels 3FN está de hecho en FNBC. I

Observe que —dadas ciertas restricciones semánticas adicionales (razonables)—esta colección de varrels FNBC es **fuertemente redundante** [5.1], en el sentido de que la proyección de lava PROY3 sobre {PROY#,DEPTO#) es en todo momento igual a una proyección de la juntadeEMP3 y TEL3 y OFICINA3.

Por último, observe que es posible "identificar" las varrels FNBC a partir del diagrama (¿cómo?). *Nota:* No afirmamos que *siempre* sea posible "identificar" una descomposición FNBC, **sólc** que a menudo es posible hacerlo en casos prácticos. Una declaración más precisa es la siguiente: di una varrel *R* que satisface un conjunto de DFs *S*, está garantizado que el algoritmo que sigue (paso 0 al 8) produce una descomposición *D* de *R* en varrels 3FN (en vez de FNBC) que son tanto sin pérdida como conservadoras de la dependencia:

0. Iniciar *D* al conjunto vacío.
1. Sea / una cobertura irreducible de 5.
2. Sea *X* un conjunto de atributos que aparecen en la parte izquierda de alguna DF *X* -> Ken/.
3. Sea *X* —>*Yl, X* —>*Y2, ... X* —>*Yn* el conjunto completo de DFs en / con la parte izquierda *X*.
4. Sea Zla unión de *Yl, Y2, ..., Yn*.
5. Reemplazar *D* por la unión de *D* y la proyección de *R* sobre *X* y Z
6. Repetir los pasos 3 al 5 para cada *X* distinta.
7. Sean *A1, A2,..., An* aquellos atributos de *R* (si los hay) aún no considerados (es decir, aún no in cluidos en ninguna varrel en *D).* Reemplazar *D* por la unión de *D* y la proyección de *R* sobreA/, *A2, ...,An*.
8. Si ninguna varrel en *D* incluye una clave candidata de *R,* reemplazar *D* por la unión de *D* y la proyección de *R* sobre alguna clave candidata de *R*.

Y está garantizado que el siguiente algoritmo (pasos 0 al 3), produce una descomposición *D*de *R* en varrels FNBC que son sin pérdida aunque no necesariamente conservan la dependencia:

0. Iniciar *D* para que contenga sólo *R.*
1. Para cada varrel *TenD* que no esté en FNBC, ejecutar los pasos 2 y 3.
2. Sea *X* —> *Y* una DF de *T* que viola los requerimientos para FNBC.
3. Reemplazar *T* en *D* por dos de sus proyecciones; o sea, la proyección sobre *X* y *Y* y la proyec ción sobre todos los atributos excepto los de *Y*.

Para volver al ejemplo de la base de datos de la compañía: como ejercicio adicional (que no tiene mucho que ver con la normalización como tal, pero que es muy importante para el diseño de bases de datos en general) trate de ampliar el diseño anterior para incorporar también las especificaciones necesarias de *claves externas.*

11.4 La figura 11.21 muestra las DFs más importantes para este ejercicio. Las suposiciones semánticas son como sigue:

- Dos clientes no pueden tener la misma dirección de envío.
- Cada pedido está identificado mediante un número de pedido único.
- Cada línea de detalle dentro de un pedido está identificada por un número de línea, único dentro del pedido.

Un conjunto adecuado de varrels FNBC es como sigue:

```
CLIENTE { CLIENTE#, EDOCTA, LIMCRED, DESCUENTO }
        KEY { CLIENTE* }

ENVIARA { DIRECCIÓN, CLIENTE* } KEY
        { DIRECCIÓN }

ENCABPEDIDO { PEDIDO*, DIRECCIÓN, FECHA }
            KEY { PEDIDO* }

LINPEDIDO { PEDIDO*, LINEA*, ART*, CANTSOL, CANTSURT }
 KEY { PEDIDO*, LINEA* } ARTICULO {
ART#, DSCTO } KEY { ART* }

ARTP { ART*, PLANTA*, EXISTENCIA, RIESGO }
     KEY { ART*, PLANTA* }
```

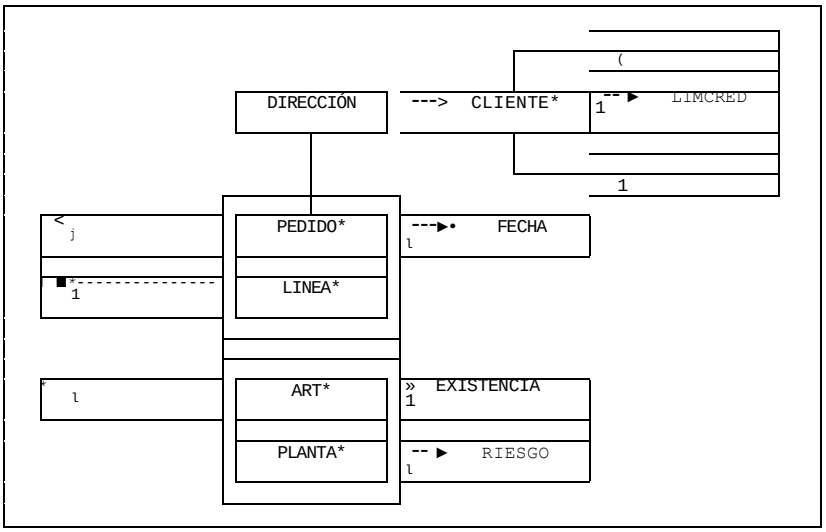


Figura 11.21 Diagrama de DFs del ejercicio **11.4.**

11.5 Considere el procesamiento que debe realizar un programa que maneja pedidos. Suponemos' el pedido de entrada especifica el número de cliente, la dirección de envío y los detalles de los art los solicitados (números de artículo y cantidades).

```
RETRIEVE CLIENTE WHERE CLIENTE* = CLIENTE# de entrada ;
verificar estado de cuenta, limite de crédito, etcétera ;
RETRIEVE ENVIARA WHERE DIRECCIÓN = DIRECCIÓN de entrada
AND    CLIENTE#   = CLIENTE* de entrada /* esto verifica la
dirección de envió */   ; IF todo está correcto THEN
procesar el pedido  ;  END IF ;
```

Si el 99 por ciento de los clientes tienen de hecho una sola dirección de envío, sería más bien in ficiente poner esa dirección en otra varrel que no fuera CLIENTE (si consideramos sólo ese 99 ciento, DIRECCIÓN es de hecho dependiente funcionalmente de CLIENTE*). Podemos mejorarla: cosas de la siguiente forma. Para cada cliente designamos una dirección de envío válida como rección *principal* de ese cliente. Por supuesto, para el 99 por ciento, la dirección principal es la única dirección. Nos referiremos a cualquier otra dirección como *secundaria*. La varrel CLIENTE puede redefinirse entonces como

```
CLIENTE { CLIENTE», DIRECCIÓN, EDOCTA, LIMCRED, DESCUENTO }
        KEY { CLIENTE* }
```

y la varrel ENVIARA puede ser reemplazada por

```
SEGUNDA { DIRECCIÓN, CLIENTE* }
        KEY { DIRECCIÓN }
```

Aquí, CLIENTE contiene la dirección principal y SEGUNDA todas las direcciones secundaria (y los números de cliente correspondientes). Ambas varrels están en FNBC. Ahora, el programa de procesamiento de pedidos luce como sigue:

```
RETRIEVE CLIENTE WHERE CLIENTE* = CLIENTE* de entrada  ;
verificar estado de cuenta, limite de crédito, etcétera  ;
IF DIRECCIÓN recuperada * DIRECCIÓN de entrada THEN
    RETRIEVE SEGUNDA WHERE DIRECCIÓN = DIRECCIÓN de entrada
                    AND    CLIENTE*     = CLIENTE* de entrada
    /* esto verifica la dirección de envío */   ; END IF ; IF
todo está correcto THEN procesar el pedido  ;  END IF ;
```

Las ventajas de este enfoque comprenden las siguientes:

- El procesamiento es más sencillo (y posiblemente más eficiente) para el 99 por ciento de los clientes.
- Si omitimos la dirección de envío en el pedido de entrada, podríamos usar en forma predeter minada la dirección principal.
- Suponga que el cliente puede tener un descuento diferente para cada dirección de envío. Con el en foque original (que mostramos como la respuesta al ejercicio previo), el atributo DESCUENTO tendría que haber sido movido a la varrel ENVIARA, haciendo aún más complicado el pro» samiento. Sin embargo, con el enfoque modificado, el descuento principal (correspondiente a dirección principal) puede ser representado por una aparición de DESCUENTO en CLIENTE, los descuentos secundarios por una correspondiente aparición de DESCUENTO en SEGUNDA Ambas varrels siguen estando en FNBC y el procesamiento es de nuevo más sencillo para el! por ciento de los clientes.

Para resumir: el aislamiento de los casos excepcionales parece ser una técnica valiosa para obtener lo mejor de ambos mundos; es decir, combinar las ventajas de la FNBC con la simplificación en la recuperación que puede ocurrir si se violan las restricciones de la FNBC.

11.6 La figura 11.22 muestra las dependencias funcionales más importantes. Una colección posible de varrels es:

```
HORARIO { L, P, C, D, H }
        KEY { L }
        KEY { P, D, H }
        KEY { C, D, H }

ESTUDIO { E, L }
        KEY { E, L }
```

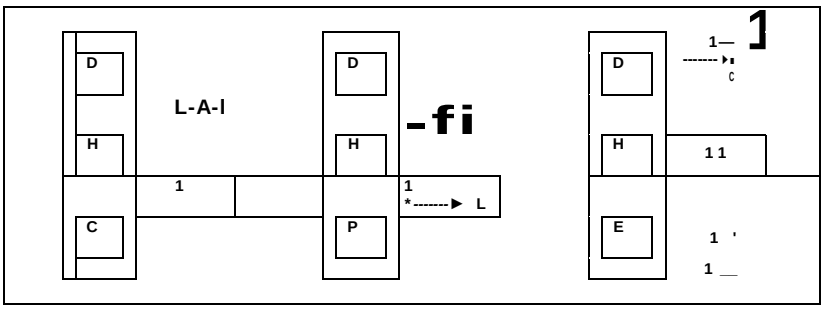


Figura 11.22 Diagrama de DFs del ejercicio 11.6.

11.7 NDIR está en 2FN pero no en 3FN (y por lo tanto, tampoco en FNBC). Un mejor diseño podría ser:

```
NCCP { NOMBRE, CALLE, CP }
     KEY { NOMBRE }

CPCE { CP, CIUDAD, ESTADO }
     KEY { CP }
```

Estas dos varrels están en FNBC. Sin embargo, observe que:

» Puesto que casi invariablemente CALLE, CIUDAD y ESTADO se requieren juntos (piense en imprimir una lista de correspondencia), y puesto que los códigos postales no cambian con mucha frecuencia, podríamos argumentar que dicha descomposición difícilmente vale la pena. (En otras palabras, la normalización debería llevarse a cabo normalmente con base en las dependencias *importantes* y no necesariamente con base en *todas* las dependencias).

■ Observe en particular que recuperar la dirección completa para un NOMBRE dado requiere ahora de una junta (aunque esa junta podría ser conciliada por el usuario al definir NDIR como una vista de NCCP y CPCE). Por lo tanto, podríamos argumentar que la normalización a FNBC es *buena para actualizar pero mala para recuperar*, es decir, la redundancia que ocurre en la ausencia de una normalización completa causa en efecto problemas con la actualización, pero podría ayudar con la recuperación.* La redundancia genera dificultades cuando *no es controlada;* aunque en ciertas circunstancias, la redundancia *controlada* (es decir, la redundancia que es declarada para y manejada mediante el DBMS) podría ser aceptable.

---

*Por otra parte, dicha redundancia en realidad puede dificultar ciertas recuperaciones (es decir, puede hacer que las consultas correspondientes sean más difíciles de formular), como veremos en la sección 12.5 del siguiente capítulo.

La DF {CALLE,CIUDAD.ESTADO} -> CP no está directamente representada por este di en su lugar, tendrá que mantenerse por separado, ya sea en forma declarativa (si el DBMS « porta un lenguaje de integridad declarativa en el sentido que esbozamos en el capítulo 8)oroe diante procedimientos. De hecho, es claro que las varrels NCCP y CPCE no son *independiente* en el sentido que señala Rissanen [11.6].

CAPITULO **12**

Normalización adicional II: formas normales superiores

12.1 INTRODUCCIÓN

En el capítulo anterior explicamos las ideas de la normalización adicional e incluso la forma normal de Boyce/Codd (que es hasta donde puede llevarnos el concepto de dependencia funcional). Ahora completamos nuestra explicación examinando la **cuarta y quinta** formas normales (4FN y 5FN). Como veremos, la definición de la cuarta forma normal hace uso de una nueva clase de dependencia, llamada DMV (dependencia **muí** ti valuada); las DMVs son una generalización de las DFs. En forma similar, la definición de la quinta forma normal hace uso de otra nueva clase de dependencia, llamada DJ (dependencia **de junta);** a su vez, las DJs son una generalización de las DMVs. La sección 12.2 explica las DMVs y la 4FN, mientras que la sección 12.3 explica las DJs y la 5FN (y explica por qué en cierto sentido especial, la 5FN es la forma normal *final).* Observe que nuestras explicaciones sobre las DMVs y las DJs son deliberadamente menos formales que nuestras explicaciones de las DFs del capítulo 10; dejamos el tratamiento formal para los artículos de investigación (vea la sección "Referencias y bibliografía").

Después, la sección 12.4 examina el procedimiento completo de normalización y hace algunos comentarios adicionales al respecto. La sección 12.5 explica brevemente la noción de ¿esnormalización. La sección 12.6 describe otro principio importante de diseño llamado diseño **ortogonal.** Por último, la sección 12.7 analiza brevemente algunas tendencias posibles para la futura investigación en el campo de la normalización, y la sección 12.8 presenta un resumen.

12.2 LAS DEPENDENCIAS MULTIVALUADAS Y LA CUARTA FORMA NORMAL

Suponga que se nos da una varrel JCPT (J de "jerárquica") que contiene información acerca de cursos, profesores y textos, y en la cual los atributos correspondientes a profesores y textos son *con valor de relación* (vea un ejemplo del valor de JCPT más adelante, en la figura 12.1). Como puede ver, cada tupia JCPT consiste en un nombre de curso más una relación que contiene los nombres de los profesores, más una relación que contiene los nombres de los textos (la figura muestra dos de estas tupias). Lo que pretende indicar dicha tupia es que el curso especificado puede ser enseñado por cualquiera de los profesores especificados y que utiliza como referencias todos los textos especificados. Damos por hecho que para un curso dado, puede existir cualquier

JCPT

CURSO	PROFESORES	TEXTOS
Física	PROFESOR: Prof. Green, Prof. Brown	TEXTO: Mecánica Básica, Principios de Óptica
Matemáticas	PROFESOR: Prof. Green	TEXTO: Mecánica Básica, Análisis Vectorial, Trigonometría

Figura 12.1 Valor de ejemplo de la varrel JCPT.

número de profesores correspondientes y cualquier cantidad de textos correspondientes. Es raí también damos por hecho —¡tal vez en forma no muy realista!— que los profesores y los textos son bastante independientes entre sí; es decir, independientemente de quién imparta realmente cualquier curso ofrecido, se utilizan los mismos textos. Por último, también damos pe hecho que un profesor o un texto determinados pueden estar asociados en cualquier cantidad c cursos.

Ahora suponga que (igual que en la sección 11.6 del capítulo anterior) queremos elimir los atributos con valor de relación. Una forma de hacerlo —aunque no la forma que describinn en la respuesta al ejercicio 11.3 (una idea a la que volveremos al final de esta sección reemplazar simplemente la varrel JCPT por una varrel CPT con tres atributos *escalares* CURSO, PROFESOR y TEXTO, como indica la figura 12.2. Como puede ver en la figura, cada tupia de JCPT da lugar a *m* * *n* tupias en CPT; donde *my n* son las cardinalidades de las relaciona PROFESORES y TEXTOS en esa tupia JCPT. Observe que la varrel resultante CPT es 'toda clave" (en contraste, la única clave candidata de JCPT era solamente {CURSO}).

CPT

CURSO	PROFESOR	TEXTO
Física	Prof. Green	Mecánica Básica
Física	Prof. Green	Principios de Óptica
Física	Prof. Brown	Mecánica Básica
Física	Prof. Brown	Principios de Óptica
Matemáticas	Prof. Green	Mecánica Básica
Matemáticas	Prof. Green	Análisis vectorial
Matemáticas	Prof. Green	Trigonometría

Figura 12.2 Valor de la varrel CPT correspondiente al valor de JCPT de la figura 12

El significado de la varrel CPT es básicamente el siguiente: una tupia {CURSO:c,PROFESOR:/>,TEXTO:í} aparece en CPT si y solamente si el curso *c* puede ser impartido por el profesor/? y utiliza el texto *t* como referencia. Observe que para un curso dado, aparecen todas las combinaciones posibles de profesor y texto; es decir, CPT satisface la restricción (de varrel)

si aparecen las tupias *(c,pl,tl), (c,p2,t2)*

entonces también aparecen las tupias *(c,pl,t2), (c,p2,tJ)*

(usando una vez más nuestra forma abreviada para tupias).

Ahora bien, debe quedar claro que la varrel CPT involucra una buena cantidad de **redundancia,** la cual (como de costumbre) conduce a ciertas **anomalías de actualización.** Por ejemplo, para agregar la información de que el curso de física puede ser impartido por un nuevo profesor, es necesario insertar *dos* nuevas tupias; una para cada uno de los textos. ¿Podemos evitar estos problemas? Bueno, resulta fácil ver que:

1. Los problemas en cuestión son generados por el hecho de que los profesores y los textos son *completamente independientes entre sí;*
2. Las cosas mejorarían mucho si se descompusiera CPT en sus dos proyecciones {CURSO,PROFESOR} y {CURSO.TEXTO} respectivamente; las llamaremos CP y CT (vea la figura 12.3).

Para agregar la información de que el curso de física puede ser impartido por un nuevo profesor, ahora sólo tenemos que insertar una tupia en la varrel CP. (También observe que podemos recuperar la varrel CPT al juntar nuevamente CP y CT, de manera que la descomposición es sin pérdida.) Por lo tanto, parece razonable sugerir que debe existir una forma de "normalizar aún más" una varrel como CPT.

Nota: en este punto, usted podría objetar que, en primer lugar, la redundancia en CPT era innecesaria y por lo tanto, que también eran innecesarias las anomalías de actualización correspondientes. De manera más específica, podría sugerir que CPT no necesita incluir todas las combinaciones posibles profesor-texto para un curso determinado; por ejemplo, es obvio que dos tupias son suficientes para mostrar que el curso de física tiene dos profesores y dos textos. El problema es *¿cuáles* son esas dos tupias? Cualquier elección particular conduce a una varrel que tiene una interpretación nada obvia y un comportamiento de actualización muy extraño (¡trate

CP

CURSO	PROFESOR
Física	Prof. Green
Física	Prof. Brown
Matemáticas	Prof. Green

CT

CURSO	TEXTO
Física	Mecánica Básica
Física	Principios de Óptica
Matemáticas	Mecánica Básica
Matemáticas	Análisis Vectorial
Matemáticas	Trigonometría

Figura 12.3 Valores de las varrels CP *y* CT correspondientes con el valor de CPT de la figura 12.2.

de enunciar el predicado de dicha varrel!; es decir, trate de establecer los criterios paradeci si una actualización dada es o no una operación aceptable sobre esa varrel).

Por lo tanto, de manera informal es obvio que el diseño de CPT está mal y es mejor 1¡ composición en CP y CT. Sin embargo, el problema es que estos hechos no son *formalmei* obvios. En particular, observe que CPT no satisface en absoluto ninguna dependencia función (salvo las triviales como CURSO -> CURSO); de hecho, CPT está en FNBC, pues como ya señalamos, es toda clave; cualquier varrel que sea "toda clave" debe estar necesariamente (FNBC. (Observe que las dos proyecciones CP y CT son también todas clave y en consecuencii están en FNBC.) Por lo tanto las ideas del capítulo anterior no ayudan en nada al problema que nos ocupa.

La existencia de varrels FNBC "problemáticas" como CPT, fue reconocida en una etapi temprana y pronto se entendió la forma de tratarlas. Sin embargo, no fue sino hasta 1977 que estas ideas intuitivas se pusieron en una posición teórica sólida, cuando Fagin presentó la noción de las **dependencias mult** i valuadas, DMVs [12.13], Las dependencias multivaluadas a una generalización de las dependencias funcionales, en el sentido de que toda DF es una DMV. aunque lo opuesto no es cierto (es decir, existen DMVs que no son DFs). En el caso de la varrel CPT hay dos DMVs válidas:

```
CURSO -»-» PROFESOR
CURSO -*-» TEXTO
```

Observe las flechas dobles; la DMV A —>—> B se lee como "B es **multidependiente** de *A",* o ***i*** manera equivalente, "A **multidetermina** a B". Nos concentraremos en la primera DMV, CURSO ->-» PROFESOR. De manera intuitiva, lo que esta DMV significa es que aunque un curso no tenga un *solo* profesor correspondiente —es decir, aunque la dependencia *funcional* CURSO —> PROFESOR *no* sea válida— cada curso sí tiene un *conjunto* bien definido de pn fesores correspondientes. Aquí, por "bien definido" queremos decir (con más precisión) que pata un curso dado *c* y un texto dado *t,* el conjunto de profesores *p* que coincide con el par (c,() en CPT, depende solamente del valor de c; independientemente de qué valor particular de *i*elijamos. La segunda DMV, CURSO —>—> TEXTO, se interpreta de manera similar. Entonces aquí tenemos la definición formal:

■ **Dependencia multivaluada.** Sea *R* una varrel y sean *A, B* y *C* subconjuntos de los atributos de *R.* Entonces decimos que *B* es **multidependiente** de *A,* en símbolos

(lea "A multidetermina a B", o simplemente "A flecha doble B"), si y solamente si en todo valor válido posible de *R,* el conjunto de valores B que coinciden con un determinado par (valor *A,* valor C) depende sólo del valor de A y es independiente del valor de *C.*

Es fácil mostrar que dada la varrel *R{A,B,C),* la DMV A —>—> B es válida si y solamente si también es válida la DMV A —>—>C; vea Fagin [12.13]. Las DMVs siempre van en pares, de esta forma. Por esta razón, es común representar ambas en un solo enunciado de esta manera:

B **i** *C*

Por ejemplo:

CURSO -*-> PROFESOR I TEXTO

Ahora bien, anteriormente dijimos que las dependencias multivaluadas son una generalización de las dependencias funcionales, en el sentido de que toda DF es una DMV. Para ser más precisos, una DF es una DMV en la cual el conjunto de valores dependientes (los de la parte derecha) que coinciden con el valor de un determinante dado (de la parte izquierda), es siempre un conjunto individual. Por lo tanto, si *A* —> *B,* entonces es cierto que *A* —>—> *B.*

Para retomar nuestro problema original de CPT, podemos ver ahora que el problema con varrels como CPT es que comprenden DMVs que no son también DFs. (En caso de que no sea obvio, señalamos que es precisamente la existencia de esas DMVs la que conduce a la necesidad de, por ejemplo, insertar *dos* tupias para agregar otro profesor de física. Esas dos tupias son necesarias para mantener la restricción de integridad representada por la DMV.) Las dos proyecciones CP y CT no involucran a ninguna de estas DMVs y es por esto que representan una mejora sobre el diseño original. Por lo tanto, quisiéramos reemplazar CPT por esas dos proyecciones, y un teorema importante demostrado por Fagin en la referencia [12.13] nos permite hacer exactamente esa sustitución:

- **Teorema** (Fagin). Sea *R{A,B, C]* una varrel, donde *A, B* y *C* son conjuntos de atributos. En tonces *R* es igual a la junta de sus proyecciones sobre *{A,B}* y *{A,C}* si y solamente si *R* satisface las DMVs *A* -*-> *B* I C.

(Observe que ésta es una versión más sólida del teorema de Heath como lo definimos en el capítulo 11). De acuerdo con Fagin [12.13], ahora definimos la *cuarta forma normal* (así llamada porque, como señalamos en el capítulo 11, la FNBC aún se denominaba en ese tiempo *tercera* forma normal):

- **Cuarta forma normal.** La varrel *R* está en 4FN si y solamente si siempre que existan subconjuntos *A* y *B* de los atributos de *R,* tales que la DMV no trivial * *A* —>—> *B* se satisfaga, entonces todos los atributos de *R* son también dependientes *funcionalmente* de *A.*

En otras palabras, las únicas dependencias no triviales (DFs o DMVs) en *R* son de la forma *K* -> *X* (es decir, una dependencia/««c¡ona/ de una superclave *K* a algún otro atributo *X).* De manera equivalente, *R* está en 4FN si y solamente si está en FNBC y todas las DMVs en *R* son de hecho "DFs sin claves". Por lo tanto, observe en particular que la 4FN implica la FNBC.

La varrel CPT no está en 4FN, ya que involucra a una DMV que no es una DF, ni siquiera una DF "sin una clave". Sin embargo, las dos proyecciones CP y CT están en 4FN. De esta manera, la 4FN es una mejora sobre la FNBC, en el sentido de que elimina otra forma de dependencia indeseable. Lo que es más, en la referencia [12.13] Fagin muestra que siempre es posible lograr la 4FN; es decir, cualquier varrel puede ser descompuesta sin pérdida en una colección equivalente de varrels 4FN; aunque nuestra explicación del ejemplo EMP de la sección 11.5 muestra que en algunos casos podría no ser bueno llevar tan lejos la descomposición (o incluso tan lejos como FNBC).

---

*Una **DMV *A* —>—> *B* es trivial** si *A* es un superconjunto de *B* o la unión de *A* y *B* es el encabezado completo.

Nota: observamos que el trabajo de Rissanen sobre proyecciones independientes [11.6], aunque asentado en términos de DFs, es aplicable también a las DMVs. Recuerde que una vam *R{A,B,C]* que satisface las DFs A —» B y B —> C se descompone mejor en sus proyecciones sobn *{A,B}* y *{B,C}* que sobre *{A,B}* y {*A,C).* Esto mismo es válido si reemplazamos las DFs por 1¡ DMVs *A* ->-> *B* y *B* -»-» C.

Concluimos esta sección retomando, como prometimos, la cuestión de eliminar atributos con valor de relación (AVRs, para abreviar) para el procedimiento específico de realizar 1¡ eliminación como lo describimos en la respuesta al ejercicio 11.3 del capítulo anterior. La idea es ésta: todo lo que necesitamos hacer en la práctica para lograr la 4FN es *reconocer que s comenzamos con una varrel que involucra dos o más AVRs independientes, lo primero qu que hacer es separar esos AVRs.* Esta regla no sólo tiene sentido intuitivo, sino que fue exactamente lo que hicimos en nuestra respuesta al ejercicio 11.3. Por ejemplo, en el caso de la varrel JCPT, lo primero que debemos hacer es reemplazar la varrel original por sus dos proye^ JCP {CURSO.PROFESORES} y JCT {CURSO,TEXTOS} (donde PROFESORES y TEXTO siguen siendo AVRs). Podemos eliminar los AVRs en esas dos proyecciones (y reducir 1; proyecciones a FNBC) de la forma usual y nunca surgirá la varrel FNBC "problemática" CPT. Pero la teoría de las DMVs y la 4FN nos dan una base formal para lo que de otro modo sería una mera regla empírica.

12.3 LAS DEPENDENCIAS DE JUNTA Y LA QUINTA FORMA NORMAL

Hasta ahora, en este capítulo (y a lo largo del capítulo anterior) hemos dado por hecho de manera tácita que la única operación necesaria o disponible en el proceso de normalización adicional es la sustitución de una varrel a una forma sin pérdida mediante *sólo dos* de sus proyecciones Esta suposición nos ha llevado con éxito hasta la 4FN. Por lo tanto, quizás resulte una sorpresa descubrir que existen varrels que no pueden ser descompuestas sin pérdida en dos proyecciones. pero que *sí pueden* hacerlo en tres (o más). Si utilizamos un término poco ortodoxo pero conveniente, describimos a dicha varrel como "descomponible en *n"* (para alguna $n > 2$); lo que significa que la varrel en cuestión puede ser descompuesta sin pérdida en *n* proyecciones pero no en *m* para cualquier $m < n$. A una varrel que puede ser descompuesta sin pérdida en dos proyecciones, la llamaremos "descomponible en 2". *Nota:* el fenómeno de la descomposición ennpai $n > 2$, fue observado primero por Aho, Beeri y Ullman [12.1]. El caso particular en que *n*» también fue estudiado por Nicolas [12.25].

Considere la varrel VPY de la base de datos de proveedores, partes y proyectos (aunqu para simplificar, ignore el atributo CANT). La parte superior de la figura 12.4 presenta un val de ejemplo. Observe que la varrel VPY es toda clave y no comprende en absoluto DFs o DMVs no triviales (y está por lo tanto en 4FN). Observe también que la figura 12.4 muestra:

a. Las tres proyecciones binarias VP, PY y YV correspondientes al valor de relación VPY(j muestra la parte superior de la figura;
b. El efecto de juntar las proyecciones VP y PY (sobre P#);
c. El efecto de juntar ese resultado y la proyección YV (sobre Y# y V#).

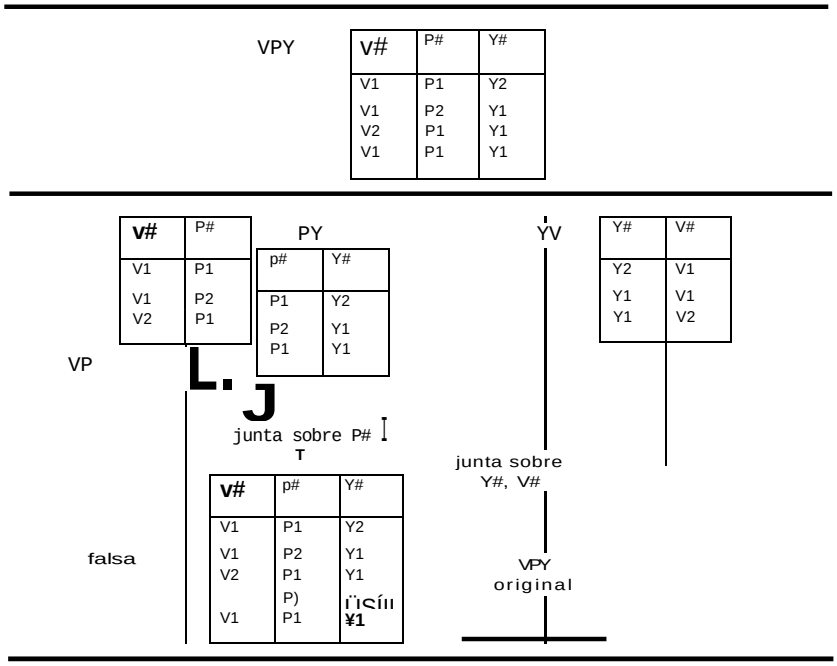


Figura 12.4 La relación VPY es la junta de sus tres proyecciones binarias, pero no de dos de ellas cualquiera.

Observe que el resultado de la primera junta es producir una copia de la relación VPY original más una tupia (falsa) adicional; y entonces el efecto de la segunda junta es eliminar esa tupia falsa, regresando así a la relación VPY original. En otras palabras, la relación VPY original es descomponible en 3. *Nota:* el resultado neto es el mismo —independientemente del par de proyecciones que elijamos para la primera junta— aunque el resultado intermedio es diferente en cada caso. *Ejercicio:* compruebe esta afirmación.

Ahora bien, el ejemplo de la figura 12.4 está expresado (por supuesto) en términos de *relaciones,* no de *varrels.* Sin embargo, sí la varrel satisface cierta restricción de integridad independiente del tiempo, la posibilidad de descomponer VPY en 3 podría ser una propiedad más fundamental independiente del tiempo; es decir, una propiedad satisfecha por todos los valores válidos de la varrel. Para entender cuál debe ser esa restricción, observe primero que el enunciado "VPY es igual a la junta de sus tres proyecciones, VP, PY y YV" equivale precisamente a la siguiente declaración:

SI	el par	***(vl,pl)***	aparece en
y	el par	VP	
y	el par	***(pl,yl)***	aparece en
entonces	la terna	PY	

(yi,W) aparece en YV
(\l,pl,yl) aparece en VPY

ya que la terna *(vl,pl,yl)* aparece obviamente en la junta de VP, PY y YV. (La declaración contraria a ésta —si *(vl.pl.yl)* aparece en VPY entonces *(vl,pl)* aparece en la proyección VP. etcétera— es claramente cierta para cualquier relación VPY de grado 3.) Puesto que *(\'¡,p¡)* aparece en VP si y solamente si *(vl,pl,y2)* aparece en VPY para cierto *y2,* y en forma sin para *(pl,yl)* y *iyl,vl),* podemos reescribir la declaración anterior como una restricción i **VPY:**

si *(vl,pl,y2), (v2,pl,yl), (vl,p2,yl)* aparece en **VPY**
entonces *(vl,pl,yl)* aparece también en VPY

Y si esta declaración es cierta en todo momento —es decir, para todos los posibles valores válidos de la varrel VPY— entonces tenemos una restricción independiente del tiempo sobre la vam (aunque resulta más bien extraña). Observe la **naturaleza cíclica** de esa restricción ("si enlazado a *pl y pl* está enlazada a *yl* y *yl* está enlazado de nuevo a *vi,* entonces *vi,pl* y vi deben coexistir en la misma tupia"). *Una varrel será descomponible en n, para alguna n* > 2, ***si*** *y solamente si satisface dicha restricción cíclica (deforma n).*

Entonces, suponga que de hecho la varrel VPY satisface esa restricción independiente tiempo (los valores de ejemplo de la figura 12.4 son consistentes con esta hipótesis). Paraabí viar, acordemos en llamar a esa restricción *Restricción D3* (D3 por descomponible en 3). ¿Qué significa la Restricción D3 en términos reales? Tratemos de ser un poco más concretos dando un ejemplo. La restricción dice que en la realidad que la varrel VPY supone representar, es un hecho que *si* (por ejemplo)

a. Smith suministra llaves inglesas, y
b. En el proyecto Manhattan se usan llaves inglesas, y
c. Smith suministra al proyecto Manhattan,

entonces

d. Smith suministra llaves inglesas al proyecto Manhattan.

Observe que (como señalamos en el capítulo 1, sección 1.3) a., b. y c. juntas *no* implican normalmente a d.; de hecho, utilizamos exactamente este ejemplo en el capítulo 1 para ilustrar "la trampa de conexión". Sin embargo, en el caso que nos ocupa decimos que *no hay trampa,* ya que existe en efecto una restricción adicional real; es decir, la Restricción D3, que hace válida en este caso específico la inferencia de d. a partir de a., b. y c.

Para retomar el tema principal de la explicación: debido a que la Restricción D3 es satisfecha si y solamente si la varrel respectiva es igual a la junta de algunas de sus proyecciones nos referimos a esa restricción como **dependencia de junta** (DJ). Una DJ es una restricción sobre la varrel respectiva tal como una DMV o una DF es una restricción sobre la varrel respectiva. Aquí está la definición:

■ **Dependencia de junta.** Sea *R* una varrel y sean *A, B,* ..., Z subconjuntos de los atributos de *R.* Entonces decimos que *R* satisface la DJ

• **{ *A*, 8, ... Z }**

(lea "estrella *A, B,* ..., Z") sí y solamente si todo valor válido posible de *R* es igual a la junta de sus proyecciones sobre A, *B,* ..., *Z.*

Por ejemplo, si acordamos usar VP para representar el subconjunto {V#,P#} del conjunto de atributos de VPY, y de manera similar para PY y YV, entonces la varrel VPY satisface la DJ » {VP,PY,YV}.

Hemos visto entonces que la varrel VPY —con su DJ » {VP,PY,YV}— puede ser descompuesta en 3. La pregunta es *¿debe* hacerse? Y la respuesta es "probablemente sf'. La varrel VPY —con su DJ— padece diversos problemas sobre las operaciones de actualización, problemas que desaparecen cuando se descompone en 3. La figura 12.5 ilustra algunos de esos problemas. Dejamos como ejercicio considerar lo que sucede después de una descomposición en 3.

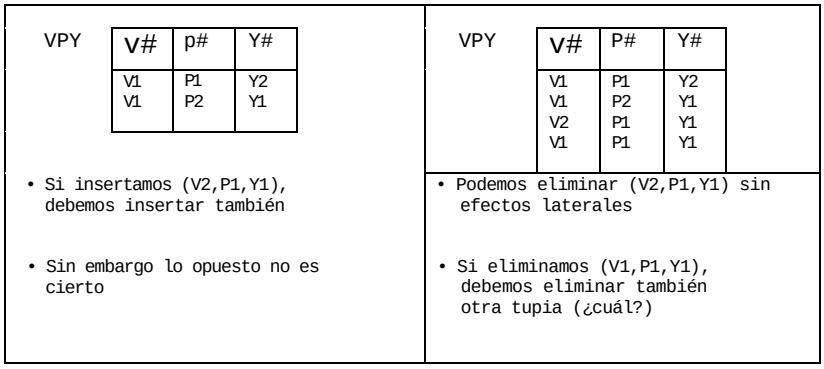


Figura 12.5 Ejemplos de problemas de actualización en VPY.

Para que *R{A,B,C]* pueda descomponerse sin pérdida en sus proyecciones sobre *{A,B}* y *{A,C}* si y sólo si las DMVs *A* —>-> *By A* -»-» *C* son válidas en *R*, el teorema de Fagin (que explicamos en la sección 12.2) puede ahora ser enunciado como sigue:

■ *R{A,B,C)* satisface la DJ » *{AB,AC}* si y sólo si satisface las DMVs *A* -»-» *B* I C.

Puesto que podemos tomar este teorema como una *definición* de dependencia multivaluada, deducimos que una DMV es sólo un caso especial de una DJ; o (de manera equivalente) que las DJs son una generalización de las DMVs. De manera formal tenemos

4+)íICs» {AB, AC}

Nota: de la definición podemos deducir que las dependencias de junta son **la forma más general posible de dependencia** (por supuesto, usamos el término "dependencia" en un sentido muy especial). Es decir, no existe una forma de dependencia aún más elevada que permita que las DJs sean simplemente un caso especial de esa forma superior, en tanto restrinjamos nuestra atención a las dependencias que tienen que ver con una varrel que se descompone mediante una proyección y se recompone mediante una junta. (Sin embargo, si permitimos otros operadores de descomposición y recomposición, entonces podrían entrar enjuego otros tipos de dependencias. Esta posibilidad la explicamos brevemente en la sección 12.7.)

Ahora bien, si retomamos nuestro ejemplo podemos ver que el problema con la varrel VPY es que involucra una DJ que no es una DMV, y por lo tanto tampoco una DF. *(Ejercicio:* ¿exactamente *por qué* se da este problema?) También vimos que es posible —y tal vez necesario— descomponer dicha varrel en componentes más pequeños; es decir, en las proyecciones especificadas mediante la dependencia de junta. Ese proceso de descomposición puede ser repetido hasta que todas las varrels resultantes estén en la *quinta forma normal,* la cual definimos ahora

- **Quinta forma normal.** Una varrel *R* está en 5FN —también llamada forma normal de **proyección-junta** (FN/PJ)— si y solamente si cada dependencia de junta no trivial* válida para *R* está implicada por las claves candidatas de *R.*

Nota: más adelante explicamos lo que significa que una DJ "esté implicada por las claves candidatas".

La varrel VPY no está en 5FN, aunque sí satisface una cierta dependencia de junta; es decir, la Restricción D3 que ciertamente no está implicada por su única clave candidata (la clave qu es la composición de todos sus atributos). Por decirlo de otra forma, la varrel VPY no está ei 5FN, ya que (a) *puede* ser descompuesta en 3 y (b) esa característica de descomposición en 3 está implicada por el hecho de que la combinación {V#,P#, Y#} es una clave candidata. En contraste, después de la descomposición en 3, las tres proyecciones VP, PY y YV están cada una en 5FN, ya que no involucran ninguna DJ (no trivial).

Aunque todavía podría no ser obvio —debido a que aún no explicamos lo que significa que una DJ esté implicada por claves candidatas— es un hecho que toda varrel en 5FN está tambi automáticamente en 4FN, ya que (como hemos visto) una DMV es un caso especial de una DJ. De hecho, en la referencia [12.14], Fagin muestra que cualquier DMV que esté implicada por una clave candidata, debe ser en realidad una DF en la que la clave candidata es el determinante. En la misma referencia [12.14], Fagin muestra además que cualquier varrel dada puede ser descompuesta en una colección equivalente de varrels 5FN; es decir, siempre es posible lograr la 5FN.

Ahora explicaremos lo que significa que una DJ esté implicada por claves candidatas Primero consideremos un ejemplo sencillo. Suponga, una vez más (como hicimos en el capítulo 11, sección 11.5), que nuestra conocida varrel de proveedores V tiene dos claves candidatas. (V#} y {PROVEEDOR}. Entonces, esa varrel satisface varias dependencias de junta; por ejemplo, satisface la DJ

```
» { { V#, PROVEEDOR, STATUS }, { V#, CIUDAD } }
```

Es decir, la varrel V es igual a la junta de sus proyecciones sobre {V#,PROVEEDOR,STATUS) y {V#,CIUDAD); de ahí que pueda ser descompuesta sin pérdida en dichas proyecciones. (Po supuesto, este hecho no significa que *deba* descomponerse así, sólo que *podría* hacerse,) Esta DJ está implicada por el hecho de que {V#} es una clave candidata (de hecho, está implicada por el teorema de Heath [11.4]). En forma similar, la varrel V satisface también la DJ

```
» { { V#, PROVEEDOR}, { V#, STATUS }, { PROVEEDOR, CIUDAD } }
```

Esta DJ está implicada por el hecho de que {V#} y {PROVEEDOR} son *ambas* claves candidatas.

Como sugiere el ejemplo anterior, una DJ » *{A,B,...,Z}* dada está implicada por las claves candidatas si *y sólo si cada una de dichas A, B, ..., Zes en realidad una superclave para la vaml*

---

*LaDJ » *{A,B,...,Z}* es **trivial** si y solamente si una de las proyecciones *A, B,* ..., Zes la proyección identidad de *R* (es decir, la proyección sobre todos los atributos de *R).*

en cuestión. Por lo tanto, dada una varrel *R,* podemos decir si está en 5FN en tanto conozcamos todas las claves candidatas y **todas las DJs** en *R.* Sin embargo, descubrir todas las DJs podría ser una operación por sí misma no trivial. Es decir, mientras que es relativamente fácil identificar las DFs y las DMVs (debido a que tienen una interpretación bastante directa en la realidad), no podemos decir lo mismo de las DJs —es decir, de las DJs que no son DMVs ni DFs— ya que el significado intuitivo de las DJs podría no ser obvio. De ahí que *aún no sea claro* el proceso de determinar cuándo una varrel dada está en 4FN pero no en 5FN y por lo tanto, determinar cuándo puede ser descompuesta para mejorar. La experiencia sugiere que dichas varrels son casos patológicos y es probable que sean raros en la práctica.

En conclusión, señalamos que a partir de esta definición, podemos deducir que la 5FN es la **última forma normal** con respecto a la proyección y la junta (que es de donde viene su nombre alterno, forma normal *de proyección-junta).* Es decir, está garantizado que una varrel en 5FN **está libre de anomalías** que pueden ser eliminadas mediante proyecciones.* Si una varrel está en 5FN, las únicas dependencias de junta son aquellas que están implicadas por claves candidatas, y por lo tanto, las únicas descomposiciones válidas son las que están basadas en esas claves candidatas. (Cada proyección de dicha descomposición consistirá en una o más de esas claves candidatas, más cero o más atributos adicionales.) Por ejemplo, la varrel de proveedores V está en 5FN. Como vimos anteriormente, *es posible* hacer descomposiciones adicionales de varias formas sin pérdida, pero toda proyección en cualquiera de estas descomposiciones seguirá incluyendo una de las claves candidatas originales (y por lo tanto no parece haber ninguna ventaja particular en esa reducción adicional).

12.4 EL PROCESO DE NORMALIZACIÓN RESUMIDO

Hasta este punto del presente capítulo (y a lo largo de todo el anterior), nos hemos estado ocupando de la técnica de *descomposición sin pérdida* como un auxiliar en el diseño de bases de datos. La idea básica es la siguiente: dada cierta varrel *R* en 1FN y algún conjunto de DFs, DMVs y DJs que se apliquen a *R,* reducimos sistemáticamente a i? en una colección de varrels "más pequeñas" (es decir, de menor grado) que son equivalentes a *R* (en cierto sentido bien definido) pero que además son en cierta forma más deseables. (La varrel original *R* podría haberse obtenido eliminando primero ciertos atributos con valor de relación, como en la sección 12.2 o —mejor aún— en la respuesta al ejercicio 11.3.) Cada paso del proceso de reducción consiste en tomar proyecciones de las varrels resultantes del paso anterior. En cada paso se usan las restricciones dadas para guiar la elección de qué proyecciones tomar después. El proceso general puede enunciarse de manera informal como un conjunto de reglas, de esta manera:

1. Tomar proyecciones de la varrel 1FN original para eliminar cualquier DF que no sea irre ducible. Este paso producirá una colección de varrels 2FN.
2. Tomar proyecciones de esas varrels 2FN para eliminar cualquier DF transitiva. Este paso producirá una colección de varrels 3FN.
3. Tomar proyecciones de dichas varrels 3FN para eliminar cualquier DF que quede y en la que el determinante no sea una clave candidata. Este paso producirá una colección de varrels

*Por supuesto, esta observación no quiere decir que esté libre de todas las anomalías posibles; sólo significa (para repetir) que está libre de las anomalías que pueden ser eliminadas al tomar proyecciones.

FNBC. *Nota:* las reglas 1 a 3 pueden resumirse en un solo lineamiento: "tomar proyecciones de la varrel original para eliminar todas las DFs en las que el determinante no sea una clave candidata".

4. Tomar proyecciones de esas varrels FNBC para eliminar cualquier DMV que no sea tai poco DF. Este paso producirá una colección de varrels 4FN. *Nota:* En la práctica es comÉ eliminar dichas DMVs —mediante "la separación de AVRs independientes", como expli camos en el ejemplo CPT en la sección 12.2— *antes* de aplicar las reglas 1 a 3 anteriores.
5. Tomar proyecciones de esas varrels 4FN para eliminar cualquier DJ que no esté implicada por las claves candidatas, aunque tal vez deberíamos agregar "si puede encontrarlas". Este paso producirá una colección de varrels en 5FN.

Del resumen anterior surgen varias ideas:

1. Antes que nada, el proceso de tomar proyecciones en cada paso debe hacerse (por su puesto) en una forma sin pérdida y de preferencia en una forma que también preserve la dependencia.

2. Observe que (como señaló primero Fagin en la referencia [12.14]) existe un paralelismc muy atractivo entre las definiciones de FNBC, 4FN y 5FN, o sea:
   - Una varrel *R* está en FNBC si y sólo si toda DF satisfecha por *R* está implicada porlas claves candidatas de *R;*
   - Una varrel *R* está en 4FN si y sólo si toda DMV satisfecha por *R* está implicada por las claves candidatas de *R;*
   - Una varrel *R* está en 5FN si y sólo si toda DJ satisfecha por *R* está implicada por las claves candidatas de *R.*

   Las anomalías de actualización que explicamos en el capítulo 11 y en secciones anterior! de este capítulo, son precisamente las anomalías que ocasionan las DFs, DMVs o DJs que no están implicadas por claves candidatas.
3. Los objetivos generales del proceso de normalización son los siguientes:
   - Eliminar cierta clase de redundancia;
   - Evitar ciertas anomalías de actualización;
   - Producir un diseño que sea una "buena" representación de la realidad, que sea intuitiva mente fácil de entender y que sea una buena base para el crecimiento futuro;
   - Simplificar el cumplimiento de ciertas restricciones de integridad.

   Ahondaremos un poco en el último elemento de esta lista. La idea general es que (cora mencionamos en los capítulos 8, 10 y en otras partes de este libro) *algunas restricciones® integridad implican a otras.* Como un ejemplo trivial, la restricción de que los salario deben ser mayores a $ 10,000 implica ciertamente a la restricción de que deben ser mayores que cero. Ahora bien, si la restricción *A* implica a la restricción *B,* entonces *hacer cumplir A hará cumplir B de manera automática* (incluso no será necesario declarar *B* explícita- mente, excepto quizás como un comentario). Y la normalización a 5FN nos da una forrai sencilla de hacer cumplir ciertas restricciones importantes que ocurren comúnmente; en esencia, todo lo que debemos hacer es hacer cumplir la unicidad de las claves candidatas

entonces todas las DJs (y todas las DMVs y las DFs) se harán cumplir automáticamente, debido por supuesto a que todas esas DJs (y DMVs y DFs) estarán implicadas por las claves candidatas.

4. Una vez más, subrayamos la idea de que los lineamientos de normalización son sólo lineamientos y que ocasionalmente podría haber buenas razones para no normalizar "todo". El ejemplo clásico en donde la normalización completa *podría* no ser una buena idea, lo ofrece la varrel de nombre y dirección NDIR (vea el ejercicio 11.7 del capítulo 11); aunque para ser francos, ese ejemplo no es muy convincente. Como regla empírica, es por lo regular una mala idea *no* normalizar todo.
5. También repetimos la idea del capítulo 11 de que las nociones de dependencia y normali zación adicional son de naturaleza *semántica;* se ocupan de lo que *significan* los datos. En contraste, el álgebra relacional, el cálculo relacional y los lenguajes como SQL que se ba san en esos formalismos, se ocupan sólo de los valores reales de los datos; no requieren y no pueden requerir ningún nivel de normalización en particular que no sea el primero. Los lineamientos de normalización adicional deben ser considerados principalmente como una disciplina para ayudar al diseñador de bases de datos (y de ahí al usuario); una disciplina mediante la cual el diseñador puede captar una parte, aunque sea pequeña, de la semántica de la realidad de una manera sencilla y directa.
6. Del punto anterior deducimos que las ideas de la normalización son útiles en el diseño de bases de datos, pero no son una panacea. Aquí tenemos algunas de las razones por las que no lo son (esta lista aparece desarrollada en la referencia [12.9]):
   - Es cierto que la normalización puede ayudar a hacer cumplir ciertas restricciones de in tegridad en forma muy sencilla, pero (como sabemos del capítulo 8) las DJs, DMVs y DFs no son las únicas clases de restricciones que pueden surgir en la práctica.
   - La descomposición podría no ser única (de hecho, por lo regular existirán muchas for mas de reducir una colección dada de varrels a 5FN) y hay pocos criterios objetivos para elegir entre descomposiciones alternativas.
   - La FNBC y los objetivos de conservación de la dependencia pueden estar en conflicto, como explicamos en la sección 11.5 ("el problema EMP").
   - El procedimiento de normalización elimina redundancias tomando proyecciones, aunque no todas las redundancias pueden ser eliminadas de esta manera ("el problema CPTD"; vea la nota a la referencia [12.13]).

   También debemos mencionar que las buenas metodologías del diseño de arriba hacia abajo tienden de todos modos a generar diseños completamente normalizados (vea el capítulo 13).

12.5 UNA NOTA SOBRE LA DESNORMALIZACION

Hasta este punto del presente capítulo (y a lo largo de todo el capítulo anterior) hemos dado por hecho que es deseable la normalización completa hasta llegar a 5FN. Sin embargo, en la práctica a menudo se afirma que es necesaria la "desnormalización" para lograr un buen desempeño. El argumento va más o menos así:

1. La normalización total significa muchas varrels separadas lógicamente (y aquí damos po: hecho que las varrels en cuestión son específicamente varrels base);
2. Muchas varrels separadas lógicamente significan muchos archivos almacenados que estái separados físicamente;
3. Muchos archivos almacenados separados físicamente significan una gran cantidad de op raciones de E/S.

Desde luego, de manera estricta este argumento no es válido, ya que (como señalamos en otra parte de este libro) el modelo relacional no estipula en ninguna parte que las varrels debaí asociarse una a una con los archivos almacenados. *La desnormalización, si es necesaria, debe hacerse en el nivel de archivos almacenados, no en el nivel de las varrels base.* Pero el argu mentó es (en cierto modo) válido para los productos SQL actuales, precisamente debido al gra inadecuado de separación entre esos dos niveles que encontramos en dichos productos. Por lo tanto, en esta sección veremos más de cerca la noción de "desnormalización". *Nota:* lasiguienl explicación se basa en gran medida en el material de la referencia [12.6],

¿Qué es la desnormalización?

Para repasar brevemente, **normalizar** una varrel *R* significa reemplazar a *R* por un conjunto de proyecciones (digamos) *Rl,..., Rn,* tales que —para todos los valores posibles *r* de la varrel *R*— si se juntan de nueva cuenta los valores *rl,..., rn* correspondientes de las proyecciones *Rl ..... Rn,* entonces se garantiza que el resultado de esa junta sea igual a r. El objetivo general es *reducir la redundancia,* asegurando que cada una de las proyecciones *Rl, ..., Rn* esté en el nivel ni alto posible de normalización (es decir, 5FN).

Ahora podemos definir la desnormalización como sigue. Sea *Rl .....Rn* un conjunto de varrels. Entonces **desnormalizar** esas varrels significa reemplazarlas por su junta (digamos) *R,* tal que para todos los valores posibles *rl, ..., rndeRl, ..., Rn* al proyectar el valor r correspondiente de *R* sobre los atributos de *Ri,* se garantiza que, se obtiene de nuevo *ri (i* = 1,..., *ri),* El objetivo general consiste en *aumentar la redundancia,* asegurando que *R* esté a un nivel menor de normalización que las varrels *Rl, ..., Rn.* En forma más específica, el objetivo es reduciré número de juntas que deben ser realizadas en tiempo de ejecución haciendo (en efecto) algunas de esas juntas antes de tiempo, como parte del disefto de la base de datos.

A manera de ejemplo, podríamos considerar la desnormalización de partes y envíos para producir una varrel PVC como indica la figura 12.6.* Observe que la varrel PVC está en lf pero no en 2FN.

Algunos problemas

El concepto de desnormalización padece diversos problemas bien conocidos. Uno obvio es que una vez que comenzamos a realizar la desnormalización, no está claro cuándo debemos detenernos. Con la normalización hay razones claramente lógicas para continuar hasta alcanzar lafora normal más alta posible; ¿concluimos entonces que con la desnormalización debemos proseguir

*Dados nuestros datos de ejemplo usuales, hay un problema con la desnormalización de proveedores y envíos; ya que el proveedor V5 se pierde en la junta. Por estas razones, algunas personas podrían argumentar que debemos emplear las juntas "externas" en el proceso de desnormalización. Pero las juntas externas tienen sus propios problemas, como veremos en el capítulo 18.

pvc

p#	PARTE	COLOR	PESO	CIUDAD	V#	CANT
P1 P1 P2	Tuerca Tuerca Perno	Rojo Rojo Verde	12.0 12.0 17.0	Londres Londres París	V1 V2 V1	300 300 200
P6	Engrane	Rojo	19.0	Londres	V1	100

Figura 12.6 Desnormalización de partes y envíos.

hasta alcanzar la forma normal *más baja* posible? Por supuesto que no, aunque no hay criterios *lógicos* para decidir exactamente dónde parar. En otras palabras, al seleccionar la desnormalización estamos retrocediendo de una posición que por lo menos tiene cierta ciencia sólida y una teoría lógica que la apoyan, a una que es de naturaleza puramente pragmática y subjetiva.

La segunda idea obvia es que hay redundancia y problemas de actualización, precisamente porque estamos tratando una vez más con varrels que no están completamente normalizadas en absoluto. Ya hemos expuesto ampliamente estos aspectos. Sin embargo, lo que es menos obvio es que también podrían haber problemas de *recuperación;* es decir, la desnormalización en realidad puede hacer que ciertas consultas sean más difíciles de expresar. Por ejemplo, considere la consulta "obtener el peso promedio para cada color de parte". Dado nuestro diseño usual normalizado, una formulación adecuada es:

```
SUMMARIZE P PER P { COLOR } ADD AVG ( PESO ) AS PESOPROM
```

Sin embargo, dado el diseño desnormalizado de la figura 12.6, la formulación es un poco más enredada (sin mencionar el hecho de que depende de la suposición —¡por lo general inválida!— de que cada parte tiene por lo menos un envío):

```
SUMMARIZE PVC { P#, COLOR, PESO } PER PVC { COLOR }
                          ADD AVG ( PESO ) AS PESOPROM
```

(También, observe que es probable que la última formulación tenga un peor rendimiento.) En otras palabras, la percepción común de que la desnormalización "es buena para la recuperación pero mala para la actualización" es en general incorrecta, por razones tanto de utilización como de rendimiento.

El tercer problema, y el más importante, es el siguiente (y esta idea se aplica a la desnormalización "propia" —es decir, a la desnormalización que se hace sólo al nivel físico— así como al tipo de desnormalización que en ocasiones tiene que hacerse en los productos SQL actuales): cuando decimos que la desnormalización "es buena para el rendimiento", en realidad queremos decir que es buena para *el rendimiento de aplicaciones específicas.* Cualquier diseño físico dado es necesariamente bueno para algunas aplicaciones y malo para otras (es decir, en términos de rendimiento). Por ejemplo, suponga que cada varrel base sí se asocia con un archivo almacenado físicamente, y suponga también que cada archivo almacenado consiste en una colección de registros almacenados físicamente en forma contigua (uno para cada tupia de la varrel correspondiente). Entonces:

- Suponga que representamos la junta de proveedores, envíos y partes como una varrel base y por lo tanto, como un archivo almacenado. Entonces la consulta "obtener los detalles de aquellos proveedores que suministran partes rojas" tendrá presuntamente un buen rendimiento frente a esta estructura física.

Sin embargo, la consulta "obtener los detalles de los proveedores de Londres" tendrá un *peor* rendimiento contra esta estructura física del que tendría si nos hubiésemos quedado con tres varrels base asociadas a tres archivos físicamente separados. La razón es queen último diseño, todos los registros almacenados de proveedores estarán físicamente contiguos, mientras que en el primer diseño estarán esparcidos físicamente en un área más amplia, y por lo tanto requerirán más operaciones de E/S.

Se aplican observaciones similares a cualquier otra consulta que acceda solamente a proveedores, o sólo a partes, o sólo a envíos, en lugar de realizar algún tipo de junta.

12.6 EL DISEÑO ORTOGONAL (UN TEMA INDEPENDIENTE)

En esta sección examinamos brevemente otro principio del diseño de bases de datos, uno que no forma parte de la normalización *en síp&ro* que se asemeja a ella debido a que es *científico.* Se llama ***el principio de diseño ortogonal.*** Considere la figura 12.7, la cual muestra un diseño para proveedores obviamente malo pero posible; en ese diseño, la varrel VA corresponde a **los** proveedores que están ubicados en París, la varrel VB corresponde a los proveedores que no están ubicados en París o que tienen un status mayor que 30 (es decir, de manera general, los predicados de varrel). Como la figura indica, el diseño conduce a ciertas *redundancias;* parase específicos, la tupia del proveedor V3 aparece dos veces, una en cada varrel.

```
/* proveedores en París */
```

VA

V#	PROVEEDOR	STATUS	CIUDAD
V2	Jones	10	París
V3	Blake	30	París

VB

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

J

```
/* proveedores que no están en París o con un
   status mayor que 30 */
```

Figura 12.7 Un mal diseño, aunque posible, para proveedores.

Por cierto, observe que la tupia *debe* aparecer en ambos lugares. Si por el contrario, suponemos que aparece (digamos) en VB pero no en VA. Entonces, aplicar la Suposición del Mundo Cerrado a VA nos diría que no se da el caso que el proveedor V3 esté ubicado en París. Sin embargo, VB nos dice que *sí* se da el caso que el proveedor V3 esté ubicado en París. En otras palabras, tendríamos entre las manos una contradicción y la base de datos sería inconsistente.

Por supuesto, el problema con la figura 12.7 es obvio: es precisamente el hecho de que es posible que una misma tupia aparezca en dos varrels distintas. En otras palabras, las dos varrels

tienen *significados que se traslapan,* en el sentido de que es posible que la misma tupia satisfaga los predicados de varrel de ambas. Entonces, una regla obvia es:

■ *El principio del diseño* ***ortogonal*** *(versión inicial).* Dentro de una base de datos dada, dos varrels no pueden tener significados que se traslapen.

Las ideas a destacar son:

1. Recuerde (del capítulo 9) que desde el punto de vista del usuario, *todas* las varrels son va rrels base (además de las vistas que están definidas como meras formas abreviadas). En otras palabras, el principio se aplica al diseño de todas las bases de datos "expresables", no sólo a la base de datos "real"; entra en acción una vez más *el principio de relatividad de la base de datos.* (Desde luego, se aplican también observaciones similares a los principios de normalización.)
2. Observe que no es posible que dos varrels tengan significados que se traslapen a menos que sean del mismo tipo (es decir, a menos que tengan el mismo encabezado).
3. Apegarnos al principio del diseño ortogonal implica que (por ejemplo) cuando insertamos una tupia, podemos considerar la operación como insertar una tupia *dentro de la base de datos,* en lugar de hacerlo dentro de una varrel específica; ya que habrá como máximo una varrel cuyo predicado satisfaga la nueva tupia.

   Ahora bien, es cierto que cuando insertamos una tupia, especificamos generalmente el nombre de la varrel *R* dentro de la cual vamos a insertar la tupia. Pero este hecho no invalida la idea anterior. El hecho es que el nombre *R* es en realidad sólo *una forma abreviada del predicado correspondiente,* digamos *PR.* En realidad estamos diciendo "INSERT tupia *t;* y por cierto, *t* es necesaria para satisfacer el predicado *PR".* Además, es claro que *R* podría ser una vista —tal vez definida mediante una expresión de la forma *A* UNION *B*— y como vimos en el capítulo 9, es muy necesario que el sistema sepa si la nueva tupia *irá a. A,* afio a ambas.

   De hecho, se aplican observaciones similares a la anterior para todas las operaciones, no solamente para las INSERTs; en todos los casos, los nombres de varrel sólo son en realidad una forma abreviada de los predicados de varrel. Ato *está de más hacer mucho énfasis en el punto de que son los predicados, y no los nombres, los que representan la semántica de los datos.*

Aún no terminamos con el principio del diseño ortogonal; existen refinamientos importantes que es necesario abordar. Considere la figura 12.8 que muestra otro diseño obviamente malo, pero posible, para proveedores. Aquí, las dos varrels por sí mismas no tienen significados que se traslapen, pero sus proyecciones sobre {V#,PRO VEEDOR} en realidad sí los tienen (de hecho, los

VX

V#	PROVEEDOR	STATUS
V1	Smith	20
V2	Jones	10
V3	Blake	30
V4	Clark	20
V5	Adams	30

VY

V#	PROVEEDOR	CIUDAD
V1	Smith	Londres
V2	Jones	París
V3	Blake	París
V4	Clark	Londres
V5	Adams	Atenas

Figura 12.8 Otro mal diseño, aunque posible, para proveedores.

significados de las dos proyecciones son idénticos). Como consecuencia, un intento por insertarla tupia —digamos— (V6,López) en una vista definida como la unión de esas dos proyecciones, hará que la tupia (V6,López,í) sea insertada en VX y la tupia (V6,López,c) en VY (donde *t* y *c* son los valores predeterminados aplicables). Resulta claro que necesitamos ampliar la definición del principio del diseño ortogonal para que se haga cargo de problemas como el de la figura 12.8.

- ***El principio del diseño ortogonal*** *(versión final).* Sean *Ay B* dos varrels base cualesquiera de la base de datos. Entonces, no deben existir descomposiciones sin pérdida de *A* y Ben *Al,..., AmycnBl,..., Bn* (respectivamente), tales que alguna proyección Ai en el conjunto *Al, ..., Am* y alguna proyección *Bj* en el conjunto *Bl, ..., Bn* tengan significados que se traslapen.

  Las ideas a destacar son:

1. Aquí, el término "descomposición sin pérdida" significa exactamente lo mismo de siempre; es decir, la descomposición en un conjunto de proyecciones tales que:
   - La varrel dada pueda ser reconstruida juntando de nuevo las proyecciones;
   - Ninguna de esas proyecciones sea redundante en el proceso de reconstrucción.
2. Esta versión del principio incluye a la versión original, ya que una descomposición sin pér dida que existe siempre para la varrel *R* es la proyección identidad de *R* (es decir, la pro yección sobre todos sus atributos).

Observaciones

1. Suponga que comenzamos con nuestra varrel usual de proveedores V, pero por motivos de diseño decidimos dividirla en un conjunto de restricciones. Entonces, el principio del di seño ortogonal nos dice que las restricciones en esa división deben estar todas disjuntas, en el sentido de que jamás ninguna tupia de proveedores puede aparecer en más de una de ellas. Nos referimos a dicha división como una *descomposición ortogonal. Nota:* el término *ortogonalidad* se deriva del hecho de que lo que en realidad significa el principio de diseño, es que las varrels base deben ser mutuamente independientes (sin traslapar significados). Por supuesto, el principio proviene del sentido común, pero del sentido *comúnformalizado* (como los principios de la normalización).
2. El objetivo general del diseño ortogonal es reducir la redundancia y evitar así las anomalías de actualización (de nuevo como la normalización). De hecho, complementa la normali zación en el sentido de que —en general— la normalización reduce la redundancia *dentro* de las varrels, mientras que la ortogonalidad la reduce *a través de* las varrels.
3. La ortogonalidad podría lograrse mediante el sentido común, pero en la práctica a menudo se ignora (de hecho, a veces se recomienda hacerlo). Los diseños como el siguiente —de una base de datos financiera— son demasiado comunes:

```
ACTIVIDADES_1997 { PARTIDA*,  DESCRIPCIÓN,  IMPORTE,  SALDO_NVO  }
ACTIVIDADESJ998 { PARTIDA*,  DESCRIPCIÓN,  IMPORTE,  SALD0_NV0  }
ACTIVIDADESJ999 { PARTIDA*,  DESCRIPCIÓN,  IMPORTE,  SALDOJWO  }
ACTIVIDADES_2000 { PARTIDA*,  DESCRIPCIÓN,  IMPORTE,  SALD0_NV0  }
ACTIVIDADES_2001 { PARTIDA*,  DESCRIPCIÓN,  IMPORTE,  SALD0_NV0  }
```

De hecho, codificar significados dentro de nombres —de varrels o de cualquier otra cosa— viola *el principio de información,* el cual establece (para recordarlo) que toda la información en la base de datos debe ser expresada explícitamente en términos de valores y de ninguna otra forma.

4. Si *A* y *B* son varrels base del mismo tipo, apegarse al diseño ortogonal implica que:

A UNION *B*	siempre es una unión disjunta;
A INTERSECT B	siempre es vacía;
A MINUS e	siempre es igual a *A.*

12.7 OTRAS FORMAS NORMALES

Volvamos a la normalización como tal. Recuerde (de la introducción al capítulo 11) que sí existen otras formas normales, además de las que hemos explicado hasta ahora en estos dos capítulos. El hecho es que la teoría de la normalización y de otros temas relacionados —ahora conocida normalmente como **teoría de la dependencia**— se ha convertido en un campo considerable por derecho propio, que cuenta con una amplia literatura. La investigación en el área continúa y de hecho florece. La exposición de esa investigación a cualquier nivel está fuera del alcance de este libro; usted puede encontrar un buen panorama general del campo (como era a mediados de los años ochenta) en la referencia [12.17]. Aquí mencionamos un par de aspectos específicos.

1. **Forma normal de dominio-clave.** La forma normal de dominio-clave (FN/DC) fue propuesta por Fagin en la referencia [12.15]. La FN/DC —a diferencia de las formas normales que hemos expuesto— no está definida en absoluto en términos de DFs, DMVs o DJs. En su lugar, se dice que una varrel *R* está en FN/DC si y solamente si toda restricción sobre *R* es una consecuencia lógica de las *restricciones de dominio* y de las *restricciones de clave* que se aplican a *R:*

   - Una *restricción de dominio* —como hemos usado aquí el término— es una restricción para que los valores de un atributo dado se tomen de algún dominio prescrito. (En la ter minología del capítulo 8, dicha restricción es en realidad una restricción de *atributo,* no una restricción de dominio.)
   - Una *restricción de clave* es una restricción para que un cierto atributo o combinación de atributos constituya una clave candidata.

   De este modo, hacer cumplir las restricciones sobre una varrel FN/DC es conceptualmente sencillo, ya que es suficiente con hacer cumplir solamente el "dominio" y las restricciones de clave para que todas las demás restricciones se hagan cumplir de manera automática. Observe con cuidado que aquí la frase "todas las demás restricciones" significa algo más que sólo DFs, DMVs y DJs; de hecho, significa el predicado completo de la varrel.

   En la referencia [12.15], Fagin muestra que cualquier varrel FN/DC está necesariamente en 5FN (y por lo tanto en 4FN y así sucesivamente) y de hecho también en FN(3,3); vea adelante. Sin embargo, la FN/DC no siempre puede lograrse, ni se ha respondido la pregunta de "¿exactamente cuándo *puede* lograrse?".

2. **Forma normal "restricción-unión".** Considere una vez más la varrel de proveedores V. La teoría de la normalización como la hemos descrito nos dice que la varrel V está en una

"buena" forma normal; de hecho, está en 5FN y por lo tanto se garantiza que está lil anomalías que pueden ser eliminadas mediante proyecciones. Pero ¿por qué mantener ¡ todos los proveedores en una sola varrel? ¿Qué tal un diseño en el que los proveedores di Londres se mantengan en una varrel (digamos VL), los proveedores de París en otra (d amos, VR) y así sucesivamente? En otras palabras ¿qué hay de la posibilidad de descomf ner la varrel original de proveedores mediante una **restricción** en lugar de una proyección ¿Sería un buen o un mal diseño la estructura resultante? (En realidad, de manera casi segura sería malo —vea el ejercicio 7.8 del capítulo 7— pero el punto es que la teoría clásica de la normalización como tal, no tiene absolutamente nada que decir en respuesta a dichas preguntas.)

Por lo tanto, otro camino en la investigación de la normalización consiste en examin las implicaciones de descomponer las varrels mediante alguna operación distinta a la pr yección. En el ejemplo, el operador de descomposición es, como ya mencionamos, una restricción (disjunta); el operador de recomposición correspondiente es una **unión** (disjunta Por lo tanto, podríamos construir una teoría de normalización "restricción-unión" análogí —aunque una vez más ortogonal— a la teoría de normalización proyección-junta c hemos venido explicando.* Personalmente, hasta donde sé, ninguna teoría como ésta ha funcionado en detalle, pero podemos encontrar algunas ideas iniciales en un documento de Smith [12.31], donde se define una nueva forma normal llamada "FN(3,3)". La FN(3,3)ii plica a la FNBC. Sin embargo, una varrel FN(3,3) no necesita estar en 4FN, ni una vam 4FN necesita estar en FN(3,3); por lo que esa reducción (como sugerimos antes) aFN(3.3) es ortogonal a la reducción a 4FN (y a 5FN). En las referencias [12.14] y [12.22] aparecen ideas adicionales sobre este tema.

12.8 RESUMEN

En este capítulo completamos nuestra explicación (iniciada en el capítulo 11) sobre la **normalización adicional.** Expusimos las **dependencias multivaluadas** (DMVs), que son una generalización de las dependencias funcionales, y las **dependencias de junta** (DJs), que son una generalización de las dependencias multi valuadas. De manera general:

- Una varrel *R{A,B, C]* satisface la DMV *A* ->-» *B* I *C* si y solamente si el conjunto de valo res *B* que coinciden con un par *(A,C)* dado depende sólo del valor de *A;* y en forma simi lar, para el conjunto de valores *C* que coinciden con un par *(A,B)* dado. Dicha varrel puede ser descompuesta sin pérdida en sus proyecciones sobre *{A,B} y {A,C};de* hecho, las DMVs son una condición necesaria y suficiente para que esta descomposición sea válida (teorema de Fagin).
- Una varrel *R{A,B,...,Z]* satisface la DJ » *{A,B,...,Z}* si y solamente si es igual a la junta de las proyecciones sobre *A, B, ... Z.* Dicha varrel puede (obviamente) ser descompuesta sin pérdida en esas proyecciones.

*De hecho, Fagin [12.14] denominó originalmente a la 5FN como *forma normal de proyección-junta,pt* cisamente porque era la forma normal con respecto a los operadores de proyección y junta.

Una varrel está en 4FN si las únicas DMVs que satisface son de hecho DFs fuera de superclaves. Una varrel está en 5FN —también llamada forma normal de **proyección-junta** (FN/PJ)— si y sólo si las únicas DJs que satisface son de hecho DFs fuera de superclaves (lo que significa que si la DJ es * *{A,B,...,Z},* entonces cada elemento de *A, B, ..., Z* es una superclave). La 5FN (la cual siempre puede ser lograda) es la *última forma normal* con respecto a la proyección y la junta.

También resumimos el **procedimiento de normalización;** lo presentamos como una secuencia informal de pasos y ofrecimos algunos comentarios importantes. Después describimos *el principio del diseño* ***ortogonal,*** el cual dice (de manera general) que dos varrels no pueden tener proyecciones con significados que se traslapen. Por último, mencionamos brevemente algunas *formas normales adicionales.*

En conclusión, quizás debamos destacar que la investigación en aspectos como los que hemos venido exponiendo es una actividad que vale mucho la pena. La razón es que el campo de la "normalización adicional" —o más bien, de la **teoría de la dependencia,** como se le llama más comúnmente— sí representa una parte de la *ciencia* en un campo (el diseño de bases de datos) que lamentablemente tiene aún mucho de esfuerzo artístico; es decir, es demasiado subjetivo y le faltan principios sólidos y lineamientos. De ahí que sea bienvenido cualquier éxito adicional en la investigación de la teoría de la dependencia.

EJERCICIOS

12.1 Las varrels CPT y VPY como las explicamos en el cuerpo del capítulo —vea los valores de ejemplo en las figuras 12.2 y 12.4— satisfacen una cierta DMV y una cierta DJ, respectivamente, que no estaban implicadas por las claves candidatas de la varrel en cuestión. Exprese esa DMV y esa DJ en la sintaxis del capítulo 8.

12.2 Sea *C* un cierto club, y sea la varrel *R{A,B]* tal que la tupia *(a,b)* aparezca en i? si y sólo si a y *b* son miembros de *C.* ¿Qué DFs, DMVs y DJs satisface *R?* ¿En qué forma normal está?

12.3 Una base de datos contendrá información relativa a representantes de ventas, áreas de venta y productos. Cada representante es responsable de las ventas en una o más áreas; cada área tiene uno o más representantes responsables. En forma similar, cada representante es responsable de la venta de uno o más productos, y cada producto tiene uno o más representantes responsables. Todos los productos se venden en todas las áreas; sin embargo, dos representantes no pueden vender el mismo producto en la misma área. Todos los representantes venden el mismo conjunto de productos en todas las áreas de las que son responsables. Diseñe un conjunto adecuado de varrels para estos datos.

12.4 En la respuesta al ejercicio 11.3 del capítulo 11, dimos un algoritmo para la descomposición sin pérdida de una varrel arbitraria *R* en un conjunto de varrels FNBC. Modifique ese algoritmo de mane ra que en su lugar produzca varrels 4FN.

12.5 *(Versión modificada del ejercicio 12.3)* Una base de datos contendrá información relativa a re presentantes de ventas, áreas de venta y productos. Cada representante es responsable de las ventas en una o más áreas; cada área tiene uno o más representantes responsables. En forma similar, cada representante es responsable de la venta de uno o más productos, y cada producto tiene uno o más re presentantes responsables. Por último, cada producto es vendido en una o más áreas y cada área tiene uno o más productos que son vendidos en ella. Además, si el representante *R* es responsable del área *A* y el producto *P* es vendido en el área *A* y el representante *R* es responsable del producto *P,* entonces *R* vende *P* en *A.* Diseñe un conjunto de varrels adecuado para estos datos.

REFERENCIAS Y BIBLIOGRAFÍA

12.1 A. V. Aho, C. Beeri y J. D. Ullman: "The Theory of Joins in Relational Databases", *ACM Tí 4,* No. 3 (septiembre, 1979). Publicado por primera vez en Proc. 19th IEEE Symp. on Foundations! Computer Science (octubre, 1977).

El artículo que señaló primero que podían existir varrels que no fueran iguales a la junta de ***i*** de sus proyecciones, pero sí iguales a la junta de tres o más. El objetivo principal del artículo f presentar un algoritmo —ahora llamado generalmente chase— para determinar si una DJ dada es o no una consecuencia lógica de un conjunto dado de DFs (un ejemplo del problema de **¡m plicación;** vea la referencia [12.17]). Este problema es equivalente al problema de determinar si una descomposición dada es sin pérdida, dado un cierto conjunto de DFs. El artículo tambiéi plica la cuestión de ampliar el algoritmo para tratar el caso en donde las dependencias dadas n(son DFs sino DMVs.

12.2 Catriel Beeri, Ronald Fagin y John H. Howard: "A Complete Axiomatization for Functional an Multi-Valued Dependencies", Proc. 1977 ACM SIGMOD Int. Conf. on Management of D Toronto, Canadá (agosto, 1977).

Amplía el trabajo de Armstrong [10.1] para incluir DMVs así como DFs. En particular, oftw el siguiente conjunto firme y completo de reglas de inferencia para DMVs:

1. **Complementation:** Si *A, B* y *C* en conjunto incluyen todos los atributbs de la varrel y *A* e un superconjunto definC, entonces *A* —>—> *B* si y solamente si *A* ->-> *C.*
2. **Reflexividad:** Si *B* es un subconjunto de *A* entonces *A* —>—> *B.*
3. **Aumento:** Si *A* —>—> *B* y *C* es un subconjunto de *D* entonces *AD* ->-» *BC.*
4. **Transitividad:** Si *A* -»-> *By B* -*-* *C* entonces *A* -»-» *C* - *B.*

Las siguientes reglas de inferencia adicionales (y útiles) pueden derivarse de las dadí anteriormente:

5. **Pseudotransitividad:** Si *A* ->-> *B* y *BC* ->-> *D* entonces *AC* ->-> *D* - *BC.*
6. **Unión:** Si *A* —»—> *B y A* —>—» C entonces *A* -^>^>*BC.*
1. **Descomposición:** Si *A* —>—> *BC* entonces *A* —>—> *B* n *C, A* —»—> *B-Cy A* ->-> *C-B.*

El artículo continúa para dar dos reglas más mediante las cuales se pueden inferir ciertas DF a partir de ciertas combinaciones de DFs y DMVs:

8. **Réplica:** Si *A* —> *B* entonces *A* —>—> *B.*
9. **Coalescencia:** Si *A* —>—> *By C* —>£>yDesun subconjunto de *B y B n C* está vacío, en tonces A —> *D.*

Las reglas de Armstrong (vea el capítulo 10) más las reglas 1^ y 8-9 anteriores conform; un conjunto firme y completo de reglas de inferencia para DFs y DMVs tomadas en conjunto. El artículo también deriva una regla útil más con relación a las DFs y las DMVs:

10. Si A ->-> *B y AB* -> C entonces *A* -> *C* - *B.*

12.3 Volkert Brosda y Gottfried Vossen: "Update and Retrieval Through a Universal Schema Inter face", *ACM TODS 13,* No. 4 (diciembre, 1988).

Los intentos anteriores de proporcionar una interfaz de "relación universal" (vea la referencií [12.19]) sólo abordaron las operaciones de recuperación. Este artículo propone un enfoque para tratar también con operaciones de actualización.

12.4 C. Robert Carlson y Robert S. Kaplan: "A Generalized Access Path Model and Its Application to a Relational Data Base System", Proc. 1976 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (junio, 1976).

Vea la nota a la referencia [12.19].

12.5 C. J. Date: "Will the Real Fourth Normal Form Please Stand Up?", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Para citar del resumen: "Hay varias nociones distintas en el mundo del diseño de bases de datos que reclaman el título de *cuarta forma normal* (4FN). La finalidad de este artículo es tratar de dejar en claro las cosas". Quizás debemos agregar que (por supuesto) la noción referida como 4FN en el cuerpo de este capítulo es la única 4FN verdadera... ¡Sin aceptar sustitutos!

12.6 C. J. Date: "The Normal Is So... Interesting" (en dos partes), *DBP&D 10,* Nos. 11-12 (noviembre-diciembre, 1997).

La explicación de la desnormalización de la sección 12.5 está tomada de este artículo. Vale la pena señalar los siguientes puntos adicionales:

- Incluso en una base de datos de sólo lectura, sigue siendo necesario declarar las restricciones de integridad, ya que definen el significado de los datos, y (como señalamos en la sección 12.4) la *no* desnormalización ofrece una forma simple de declarar ciertas restricciones importantes. Y si la base de datos *no* es de sólo lectura, entonces la no desnormalización proporciona una forma sencilla de *hacer cumplir* también dichas restricciones.
- La desnormalización implica aumentar la redundancia, pero (al contrario de la opinión popular) la redundancia incrementada no necesariamente implica desnormalización. Muchos autores han caído en esta trampa, y algunos siguen cayendo.
- Como regla general, la desnormalización (es decir, la desnormalización en el nivel lógico) debe intentarse como una táctica de rendimiento "sólo si falla todo lo demás" [4.16].

12.7 C. J. Date: "The Final Normal Form!" (en dos partes), *DBP&D 11,* Nos. 1-2 (enero-febrero, 1998).

Un tutorial sobre DJs y 5FN.

12.8 C. J. Date: "What's Normal, Anyway?", *DBP&D 11,* No. 3 (marzo, 1998).

Una indagación sobre ciertos ejemplos "patológicos" de normalización, como el ejemplo EUA del ejercicio 11.2 del capítulo 11.

12.9 C. J. Date: "Normalization Is No Panacea", *DBP&D 11,* No. 4 (abril, 1998).

Un estudio sobre algunos aspectos de diseño de bases de datos que *no* son apoyados por la teoría de la normalización. El artículo no está concebido como un ataque.

12.10 C. J. Date y Ronald Fagin: "Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992). También publicado en *ACM TODS 17,* No. 3 (septiembre, 1992).

Muestra que si (a) la varrel *R* está en 3FN y (b) todas las claves candidatas de *R* son simples, entonces *R* está automáticamente en 5FN. En otras palabras, en el caso de dicha varrel, no hay que preocuparse por los temas comparativamente más complicados —DMVs, DJs, 4FN, 5FN— expuestos en el presente capítulo. *Nota:* el artículo también demuestra otro resultado, para ser más precisos, que si (a) *R* está en FNBC y (b) por lo menos una de sus claves candidatas es simple, entonces *R* está automáticamente en 4FN, pero no necesariamente en 5FN.

12.11 C. J. Date y David McGoveran: "A New Database Design Principle", en C. J. Date, *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).

12.12 C. Delobel y D. S. Parker: "Functional and Multi-Valued Dependencies in a Relational D base and the Theory of Boolean Switching Functions", Tech. Report No. 142, Dept. Maths. Appl. Informatique, Univ. de Grenoble, Francia (noviembre, 1978).

Amplía los resultados de la referencia [10.3] para incluir DMVs así como DFs.

12.13 Ronald Fagin: "Multi-Valued Dependencies and a New Normal Form for Relational Data bases", *ACM TODS 2*, No. 3 (septiembre, 1977).

La nueva forma normal era 4FN.

Aquí, agregamos una nota sobre las dependencias multivaluadas **incrustadas.** Suponga q ampliamos la varrel CPT de la sección 12.2 para incluir el atributo adicional DÍAS, que represent! el número de días dedicados al TEXTO indicado por el PROFESOR indicado en el CURSO cado. Nos referiremos a esta varrel como CPTD. Aquí tenemos un valor de ejemplo:

CPTD

CURSO	PROFESOR	TEXTO	DÍAS
Física	Prof. Green	Mecánica Básica	5
Física	Prof. Green	Principios de Óptica	5
Física	Prof. Brown	Mecánica Básica	6
Física	Prof. Brown	Principios de Óptica	4
Matemáticas	Prof. Green	Mecánica Básica	3
Matemáticas	Prof. Green	Análisis Vectorial	3
Matemáticas	Prof. Green	Trigonometría	4

La combinación {CURSO,PROFESOR,TEXTO} es una clave candidata y tenemos la DF

```
{ CURSO, PROFESOR, TEXTO } -> DÍAS
```

Observe que la varrel está en la cuarta forma normal; no involucra alguna DMV que no seat; bien una DF. Sin embargo, sí incluye dos DMV *incrustadas* (de PROFESOR sobre CURSO de TEXTO sobre CURSO). Decimos que la DMV incrustada de *B* sobre *A* es válida en la varrel *R* si la DMV "regular" *A* —>—> *B* es válida en alguna proyección de *R.* Una DMV regular es i caso especial de una DMV incrustada, pero no todas las DMVs incrustadas son DMVs regulas

Como ilustra el ejemplo, las DMVs incrustadas implican redundancia, tal como las DMVs regulares; sin embargo (en general) la redundancia no puede eliminarse mediante proyecciones. La varrel mostrada arriba no puede en absoluto ser descompuesta (sin pérdida) en proyección! —de hecho, está en la quinta forma normal, así como en la cuarta— debido a que DÍAS depende de los tres: CURSO, PROFESOR y TEXTO y no puede aparecer en una varrel con ningunode los tres en absoluto. Por lo tanto, en su lugar las dos DMVs incrustadas tendrían que ser de claradas como restricciones adicionales explícitas sobre la varrel. Dejamos los detalles como ejercicio.

12.14 Ronald Fagin: "Normal Forms and Relational Database Operators", Proc. 1979 ACM SK MOD Int. Conf. on Management of Data, Boston, Mass. (mayo-junio, 1979).

Éste es el artículo que introdujo el concepto de forma normal de proyección-junta (FN/PJ o5F Sin embargo, es también mucho más que eso. Puede ser considerado como la declaración definitiva de lo que podría llamarse teoría de normalización "clásica"; es decir, la teoría de la descomposición sin pérdida basada en la proyección como el operador de descomposición y la juna natural como el operador de recomposición correspondiente.

12.15 Ronald Fagin: "A Normal Form for Relational Databases that Is Based on Domains and Keys", *ACM TODS 6*, No. 3 (septiembre, 1981).

12.16 Ronald Fagin: "Acyclic Database Schemes (of Various Degrees): A Painless Introduction", IBM Research Report RJ3800 (abril, 1983). Publicado nuevamente en G. Ausiello y M. Protasi (eds.), *Proc. CAAP83 8th Colloquium on Trees in Algebra and Programming* (Springer Verlag Lecture Notes in *Computer Science 159).* Nueva York, N.Y.: Springer Verlag (1983).

La sección 12.3 del presente capítulo mostró cómo una cierta varrel ternaria VPY que satisfizo cierta restricción cíclica podía ser descompuesta sin pérdida en sus tres proyecciones binarias. Decimos que la estructura de base de datos resultante (es decir el esquema, llamado *scheme* en este documento) es **cíclica,** debido a que cada una de las tres varrels tiene un atributo en común con cada una de las otras dos. (Si dibujamos la estructura como un *hipergrafo,* en la que los arcos representan las varrels individuales y el nodo en la intersección de dos arcos corresponde precisamente a los atributos en común para los mismos, entonces debe quedar claro el por qué se usa el término "cíclica".) En contraste, la mayoría de las estructuras presentadas en la práctica no son cíclicas. Estas últimas estructuras gozan de diversas propiedades formales que no se aplican a las estructuras cíclicas. En este artículo, Fagín presenta y explica una lista de dichas propiedades.

Una forma útil de pensar en la no ciclicidad es la siguiente: así como la teoría de la normalización puede ayudar a determinar cuándo una *varrel individual* debe ser reestructurada de alguna manera, también la teoría de la no ciclicidad puede ayudar a determinar cuándo una *colección* de varrels debe ser reestructurada de alguna manera.

12.17 R. Fagin y M. Y. Vardi: "The Theory of Data Dependencies—A Survey", IBM Research Report RJ4321 (junio, 1984). Publicado nuevamente en *Mathematics of Information Processing: Proc. Sym posia in Applied Mathematics 34,* American Mathematical Society (1986).

Ofrece una breve historia del tema de la teoría de la dependencia a mediados de los años ochenta (observe que aquí "dependencia" *no* se refiere solamente a las DFs). En particular, el artículo resume los mayores logros en tres áreas específicas dentro del campo general, y al hacerlo proporciona una buena lista selecta de referencias importantes. Las tres áreas son (1) el problema de implicación, (2) el modelo de "relación universal" y (3) los esquemas no cíclicos. El **problema de implicación** es el problema de determinar —a partir de un conjunto de dependencias *D* y de alguna dependencia específica *d*— si *des* una consecuencia lógica de *D* (vea la sección 10.7). Explicamos brevemente el **modelo de relación universal** y los **esquemas no cíclicos** en las notas a las referencias [12.19] y [12.16], respectivamente.

12.18 Ronald Fagin, Alberto O. Mendelzon y Jeffrey D. Ullman: "A Simplified Universal Relation Assumption and Its Properties", *ACM TODS 7,* No. 3 (septiembre, 1982).

Conjetura que siempre es posible representar la realidad a través de una "relación universal" [12.19] —o, más bien, mediante una varrel universal— que satisface precisamente una dependencia de junta más un conjunto de dependencias funcionales, y explora algunas de las consecuencias de esa conjetura.

12.19 W. Kent: "Consequences of Assuming a Universal Relation", *ACM TODS 6,* No. 4 (diciem bre, 1981).

El concepto de **relación universal** se manifiesta a sí mismo en varias formas diferentes. En primer lugar, la disciplina de normalización que describimos en los dos últimos capítulos supuso de manera tácita que es posible definir una *relación* universal inicial —o en forma más correcta, una varrel universal— que incluye todos los atributos relevantes de la base de datos en consideración, y después mostró cómo esa varrel puede ser sustituida por proyecciones sucesivas "más pequeñas" (de menor grado) hasta alcanzar cierta estructura "correcta". Pero ¿es realista o justificable esa suposición inicial? La referencia [12.19] sugiere que no, tanto en el terreno práctico como en el teórico. La referencia [12.32] es una respuesta a la referencia [12.19] y la referencia [12.20] es una respuesta a esa respuesta.

La segunda —y más importante pragmáticamente— manifestación del concepto de la varrel universal es una *interfaz de usuario.* Aquí la idea básica es bastante directa, y de hecho (desde un punto de vista intuitivo) bastante atractiva. Los usuarios deben tener la posibilidad de formular sus peticiones de base de datos, no en términos de varrels y juntas entre sí, sino más bien sole en términos de atributos. Por ejemplo:

```
STATUS WHERE COLOR ■ COLOR ( 'Rojo¹ )
```

("obtener el status de los proveedores que suministran alguna parte roja"). En este punto, la idea se bifurca en dos interpretaciones más o menos distintas:

1. Una posibilidad es que el sistema deba determinar por sí mismo —de alguna manera—qué rutas de acceso lógico seguir (en particular, qué juntas efectuar) con el fin de responder a la consulta. Éste es el enfoque sugerido en la referencia [12.4] (el cual parece haber sido el primer artículo en exponer la posibilidad de una interfaz de "relación universal", aunque no utilizó el término). Este enfoque depende de manera crítica de la denominación adecuada de los atributos. Así, por ejemplo, a los dos atributos de número de proveedor (en las varrels V y VP, respectivamente) se les *debe* dar el mismo nombre; en el caso contrario, a los atribu tos ciudad de proveedor y ciudad de partes (en las varrels V y P, respectivamente) *no* se les debe dar el mismo nombre. Si se viola cualquiera de estas dos reglas, habrá ciertas consultas que el sistema no podrá manejar adecuadamente.
2. El otro enfoque —menos ambicioso— consiste simplemente en ver a todas las consultas como si estuvieran formuladas en términos de un conjunto *predefinido* de juntas; en efecto. una vista predefinida consiste en "la" junta de todas las varrels de la base de datos.

Aunque no se cuestiona el hecho de que cualquiera de los dos enfoques simplificaría en gran medida la expresión de muchas consultas que se presentan en la práctica —y de hecho es esencial un enfoque así para soportar cualquier componente frontal de lenguaje natural— también queda claro que el sistema debe, en general, soportar la capacidad de especificar también rutas de acceso (lógicas). Para ver que esto debe ser así, considere la consulta

```
STATUS WHERE COLOR * COLOR ( 'Rojo' )
```

¿Esta consulta significa "obtener el status de los proveedores que suministran una parte que no es roja" o bien, "obtener el status de los proveedores que no suministran una parte roja"? Cualquiera que sea, debe haber una manera de formular la otra. (A propósito, el primer ejemplo de arriba también es susceptible a otra interpretación: "obtener el status de los proveedores que *solamente* suministran partes rojas".) Y aquí tenemos un tercer ejemplo: "obtener los pares de proveedores que estén coubicados". Aquí, de nuevo es claro que será necesaria una junta explícita (ya que, de manera general, el problema involucra una junta de la varrel V consigo misma).

12.20 William Kent: "The Universal Relation Revisited", *ACM TODS 8,* No. 4 (diciembre. 1983).

12.21 Henry F. Korth *et al.*\ "System/U: A Database System Based on the Universal Relation As sumption", *ACM TODS* 9, No. 3 (septiembre, 1984).

Describe la teoría, DDL, DML y la implementación de un sistema de "relación universal" experimental construido en la Universidad de Stanford.

12.22 David Maier y Jeffrey D. Ullman: "Fragments of Relations", Proc. 1983 SIGMOD Int. **Contal** Management of Data, San Jose, Calif, (mayo, 1983).

12.23 David Maier, Jeffrey D. Ullman y Moshe Y. Vardi: "On the Foundations of the Universal Re lation Model", *ACM TODS 9,* No. 2 (junio, 1984). Una versión anterior de este artículo apareció, bajo

el título "The Revenge of the JD", en Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Atlanta, Ga. (marzo, 1983).

12.24 David Maier y Jeffrey D. Ullman: "Maximal Objects and the Semantics of Universal Relation Databases", *ACM TODS 8*, No. 1 (marzo, 1983).

Los *objetos maximales* representan un enfoque al problema de ambigüedad que surge en los sistemas de "relación universal" cuando la estructura subyacente es cíclica (vea la referencia [12.16]). Un objeto maximal corresponde a un subconjunto predeclarado de la totalidad de atributos para los cuales la estructura subyacente *no* es cíclica. Dichos objetos son utilizados para guiar la interpretación de las consultas que de otro modo serían ambiguas.

12.25 J.-M. Nicolas: "Mutual Dependencies and Some Results on Undecomposable Relations", Proc. 4th Int. Conf. on Very Large Data Bases, Berlín, República Federal de Alemania (septiembre, 1978).

Presenta el concepto de "dependencia mutua". Una dependencia mutua es en realidad un caso especial de la dependencia de junta general —es decir, una DJ que no es una DMV ni DF— que comprende exactamente tres proyecciones (como el ejemplo de DJ que dimos en la sección 12.3). No tiene nada que ver con el concepto de dependencia mutua que explicamos en el capítulo 11.

12.26 Sylvia L. Osborn: "Towards a Universal Relation Interface", Proc. 5th Int. Conf. on Very Large Data Bases, Rio de Janeiro, Brasil (octubre, 1979).

Las propuestas de este artículo dan por hecho que si hay dos o más secuencias de juntas en un sistema de "relación universal" (que generarán una respuesta candidata a una consulta dada) entonces la respuesta deseada es la unión de todos esas candidatas. Proporciona algoritmos para generar todas esas secuencias de juntas.

12.27 D. Stott Parker y Claude Delobel: "Algorithmic Applications for a New Result on Multi-Valued Dependencies", Proc. 5th Int. Conf. on Very Large Data Bases, Río de Janeiro, Brasil (oc tubre, 1979).

Aplica los resultados de la referencia [12.12] a varios problemas, como el problema de probar una descomposición sin pérdida.

12.28 Y. Sagiv, C. Delobel, D. S. Parker y R. Fagin: "An Equivalence between Relational Database Dependencies and a Subclass of Propositional Logic", *JACM 28*, No. 3 (junio, 1981).

Combina las referencias [10.8] y [12.29].

12.29 Y. Sagiv y R. Fagin: "An Equivalence between Relational Database Dependencies and a Sub class of Propositional Logic", IBM Research Report RJ2500 (marzo, 1979).

Amplía los resultados de la referencia [10.8] para incluir tanto DMVs como DFs.

12.30 E. Sciore: "A Complete Axiomatization of Full Join Dependencies", *J ACM 29*, No. 2 (abril, 1982).

Amplía el trabajo de la referencia [12.2] para incluir tanto DJs como DFs y DMVs.

12.31 J. M. Smith: "A Normal Form for Abstract Syntax", Proc. 4th Int. Conf. on Very Large Data Bases, Berlín, República Federal de Alemania (septiembre, 1978).

El artículo que introdujo la FN(3,3).

12.32 Jeffrey D. Ullman: "On Kent's 'Consequences of Assuming a Universal Relation'", *ACM TODS 8*, No. 4 (diciembre, 1983).

12.33 Jeffrey D. Ullman: "The U.R. Strikes Back", Proc. 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, Calif, (marzo, 1982).

RESPUESTAS A EJERCICIOS SELECCIONADOS

12.1 Aquí tenemos primero la DMV para la varrel CPT:

```
CONSTRAINT DMV_CPT
   WITH
      ( CPT RENAME CURSO AS C, PROFESOR AS P, TEXTO AS T ) AS T1,
      ( EXTEND T1
        ADD ( CPT WHERE CURSO = C ) { P } AS A ) AS T2,
      ( EXTEND T2
      ADD ( CPT WHERE CURSO = C AND TEXTO = T ) { P } AS B ) AS T3, (
      T3 WHERE A * B ) AS T4 : IS_EMPTY ( T4 ) ;
```

Por supuesto, también existe una formulación mucho más sencilla:

```
CONSTRAINT DMV_CPT CPT • CPT { CURSO, PROFESOR } JOIN
                         CPT { CURSO, TEXTO } ;
```

Y aquí está la DJ para la varrel VPY:

```
CONSTRAINT DJ_VPY VPY ■ VPY { V#, P# } JOIN
                        VPY { P#, Y# } JOIN
                        VPY { Y#, V# } ;
```

12.2 Primero, observe que *R* contiene todo valor posible de *A,* junto con todo valor posible de S. y además que el conjunto de todos los valores de *A* (digamos *S)* es el mismo que el conjunto de todos los valores de *B.* Por lo tanto, el cuerpo de *R* es igual al producto cartesiano del conjunto *S* consigo mismo; de manera equivalente, *R* es igual al producto cartesiano de sus proyecciones *R{A]* y *R\B\.* De esta manera, *R* satisface las siguientes DMVs (las cuales no son triviales, ya que ciertamente no son satisfechas por todas las varrels binarias):

{ } -*y^A* I S

De manera equivalente, *R* satisface la *DJ* » *{A,B}* (recuerde que cuando no hay atributos en común. la junta degenera en producto cartesiano). Deducimos que *R* no está en 4FN y que puede ser descompuesta sin pérdida en sus proyecciones sobre A y *B* (por supuesto, esas proyecciones tienen cuerpos idénticos). Sin embargo, *R* está en FNBC (es toda clave) y no satisface DFs no triviales. *Nota: R* satisface también las DMVs

4-HBI { }

***B* -++*A* I { }**

Sin embargo, estas DMVs *son* triviales, ya que son satisfechas por toda varrel binaria con los atributos *Ay B.*

12.3 Primero introducimos tres varrels

```
REP      { REP#, ... }
          KEY { REP# }

AREA     { AREA#, . . . } KEY
           { AREA# }

PRODUCTO { PR0D#, ... } KEY
           { PR0D# }
```

con las interpretaciones obvias. Segundo, podemos representar el vínculo entre representantes de ventas y áreas de ventas mediante una varrel

```
RA { REP*, AREA* }
   KEY { REP#, AREA# }
```

y el vínculo entre representantes de ventas y productos por medio de una varrel

```
RP { REP#, PR0D# } KEY
   { REP#, PROD# }
```

(estos dos vínculos son muchos a muchos).

A continuación, se nos dice que todos los productos son vendidos en todas las áreas. Así que si introducimos una varrel

```
AP { AREA#, PROD# } KEY {
   AREA#, PR0D# }
```

para representar los vínculos entre áreas y productos, entonces tenemos la restricción (que llamaremos C) de que

```
AP = AREA { AREA* } TIMES PRODUCTO { PROD* }
```

Observe que la restricción C implica que la varrel AP no está en 4FN (vea el ejercicio 12.2). De hecho, la varrel AP no nos da ninguna información que no podamos obtener de las otras varrels; para ser más precisos, tenemos

```
AP { AREA* } = AREA { AREA* }
```

y

```
AP { PROD* } = PRODUCTO { PROD* }
```

Sin embargo, supongamos por el momento que la varrel AP *está* incluida de todos modos en nuestro diseño.

Dos representantes de ventas no pueden vender el mismo producto en la misma área. En otras palabras, dada una combinación {ÁREA#,PROD#}, hay exactamente un representante de ventas responsable (REP#), así que podemos introducir una varrel

```
APR { AREA#, PROD*, REP* }
   KEY { AREA*, PROD* }
```

en la cual (para hacer la DF explícita)

```
{ AREA*, PROD* } -» REP*
```

(por supuesto, la especificación de la combinación {AREA#,PROD#} como clave candidata es suficiente para expresar esta DF). Ahora, sin embargo, las varrels RA, RP y AP son redundantes, ya que son proyecciones de APR; por lo tanto podemos eliminarlas todas. En vez de la restricción C, ahora necesitamos una restricción Cl:

```
APR { AREA*, PROD* } = AREA { AREA* } TIMES PRODUCTO { PROD* }
```

Esta restricción debe ser declarada por separado y de manera explícita (no está "implicada por las claves candidatas").

Además, puesto que todo representante de ventas vende todos los productos que representa todas las áreas que representa, tenemos la restricción adicional C2 sobre la varrel APR:

```
REP# ->^> AREA# I PROD#
```

(una DMV no trivial; la varrel APR no está en 4FN). De nuevo, la restricción debe ser declarada pe separado y en forma explícita. De esta manera el diseño final consiste en las varrels REP, AREA. PRODUCTO y APR, junto con las restricciones explícitas Cl y C2.

Este ejercicio ilustra en forma muy clara la idea de que (en general) la disciplina de la normalización es adecuada para representar algunos aspectos semánticos de un problema dado (en ei dependencias que están implicadas por claves candidatas, donde por "dependencias" nos refei DFs, DMVs o DJs), aunque esa declaración explícita de dependencias adicionales podría también se necesaria para otros aspectos; y algunos aspectos no pueden representarse en absoluto en términos de dichas dependencias. Además ilustra (una vez más) la idea de que no siempre es necesario normalia "hasta el final" (la varrel APR está en FNBC pero no en 4FN).

12.4 La modificación es directa; todo lo que necesitamos es reemplazar las referencias a DI FNBC por referencias análogas a DMVs y 4FN, así tenemos:

0. Iniciar D para contener solamente a *R*.
1. Para cada varrel *TenD* que no esté en 4FN, ejecutar los pasos 2 y 3.
2. Sea *X* —>—> *Y* una DMV para *T* que viola los requerimientos de 4FN.
3. Reemplazar *Ten D* por dos de sus proyecciones; es decir, la proyección sobre *X y Y y* la proyec ción sobre todos los atributos con excepción de aquellos en *Y*.

12.5 Este es un ejemplo de "restricción cíclica". El siguiente diseño es adecuado:

```
REP      { REP#, ... }
          KEY { REP# }

AREA     { AREA#, . . . }
          KEY { AREA# }

PRODUCTO { PR0D#, ... }
          KEY { PR0D# }

RA { REP#, AREA# }
   KEY { REP#, AREA* }

AP { AREA#, PR0D# }
   KEY { AREA#, PR0D# }

PR { PR0D#, REP# }
   KEY { PR0D#, REP# }
```

Además, el usuario necesita estar informado de que la junta de RA, AP y PR *no* involucra una "tramp de conexión":

```
( RA JOIN AP JOIN PR ) { REP#, AREA# } = RA AND ( RA
JOIN AP JOIN PR ) { AREA#, PR0D# } = AP AND ( RA
JOIN AP JOIN PR ) { PR0D#, REP# } = PR
```

CAPITULO 13

Modelado semántico

13.1 INTRODUCCIÓN

El modelado semántico ha sido tema de investigación desde finales de los años setenta. La motivación general para esa investigación —es decir, el problema que los investigadores han tratado de resolver— es ésta: por lo regular los sistemas de bases de datos sólo tienen una comprensión muy limitada de lo que *significa* la información de la base de datos. Por lo general "entienden" ciertos valores de datos simples, y quizás ciertas restricciones simples que se aplican a dichos valores, pero prácticamente nada más (cualquier interpretación más sofisticada es dejada al usuario). Y sería bueno que los sistemas pudieran entender un poco más,* de modo que pudiesen responder de manera un poco más inteligente a las interacciones del usuario, y tal vez soportar interfaces de usuario más sofisticadas (de más alto nivel). Por ejemplo, sena bueno que SQL entendiera que los pesos de las partes y las cantidades de los envíos, aunque es obvio que ambos son valores numéricos, son de diferente clase —es decir, *semánticamente* diferentes— de manera que (por ejemplo) una solicitud para juntar partes y envíos sobre la base de pesos y cantidades coincidentes, pudiera al menos ser cuestionada, si no es que rechazada de una vez.

Por supuesto, la noción de *dominios* (o tipos) es muy importante para este ejemplo en particular (el cual sirve para ilustrar la idea importante de que los modelos de datos actuales no están desprovistos del todo de características semánticas). Por ejemplo, los dominios, las claves candidatas y las claves externas son todos características semánticas del modelo relacional, **tal** como lo definimos originalmente. Puesto de otra forma, los diversos modelos de datos "extendidos" desarrollados a través de los años para abordar el aspecto semántico, son sólo ligeramente más semánticos que los primeros modelos. Para parafrasear a Codd [13.6], captar el significado de los datos es una tarea que no termina nunca, y esperamos (¡o deseamos!) ver desarrollos continuos en esta área conforme nuestro entendimiento sigue evolucionando. Por lo tanto, el término **modelo semántico** (empleado frecuentemente para referirse a uno u otro de los modelos "extendidos") no es particularmente adecuado, ya que tiende a sugerir que el modelo en cuestión se las arregla de algún modo para captar *toda* la semántica de la situación en consideración. Por

---

*Sobra decir que nuestra postura es que un sistema que soporte **predicados de base de datos y de varrel** (como explicamos en el capítulo 8) *"entenderá* un poco más"; es decir, argumentamos que dicho soporte de predicado es la base correcta y adecuada para el modelado semántico. Sin embargo, desafortunadamente la mayoría de los esquemas de modelado semántico no están basados en ninguno de estos cimientos sólidos, sino que son *adecuados* hasta cierto punto (las propuestas de las referencias [13.17—13.19] son una excepción). Pero esta situación podría cambiar gracias a la creciente conciencia en el mundo comercial de la importancia de las **reglas** de **negocios** [8.18-8.19]; en este sentido, los predicados del capítulo 8 son básicamente "reglas de negocios".

otra parte, el término "modelado semántico" es una etiqueta apropiada para la actividad ge de intentar representar el significado. En este capítulo presentamos primero una breve introducción a las ideas que subyacen a esta actividad, y después examinamos con cierta profundida< enfoque en particular: el enfoque *entidad/vínculo* (que es el que se usa más comúnmente en práctica).

Observamos que el modelado semántico es conocido por muchos nombres, incluyendo modelado de *datos,* modelado *entidad/vínculo,* modelado de *entidades* y modelado de *objetí* Nosotros preferimos modelado *semántico* por las siguientes razones:

- No nos gusta "modelado de datos" porque (a) choca con nuestro uso del término "model de datos" que establecimos anteriormente para referirnos a un sistema formal con aspectos estructurales, de integridad y de manipulación; y (b) tiende a reforzar la concepción popu lar equivocada de que un modelo de datos (en nuestro sentido) comprende *solamente* es tructuras de datos. *Nota:* es importante recordarle la idea que señalamos en el capítulo 1 sección 1.3, con respecto a que el término *modelo de datos* se usa en la literatura con significados distintos. El primero es como un modelo de *datos en general* (en este sentic el modelo relacional es un modelo de datos). El segundo es como un modelo de los datos persistentes de *alguna empresa en particular.* Nosotros no usamos el término en este último sentido, pero usted debe saber que muchos autores lo hacen.
- Tampoco nos agrada "modelado entidad/vínculo" ya que tiende a sugerir que sólo hay un enfoque específico al problema; mientras que (por supuesto) en la práctica son posibles mu chos enfoques diferentes. Sin embargo, el término "modelado entidad/vínculo" está bien es tablecido, es muy popular y se encuentra con mucha frecuencia.
- No tenemos objeciones de fondo para "modelado de entidades", con excepción de que parece un poco más específico como etiqueta que "modelado semántico", y por lo tanto po dría sugerir un énfasis que no se pretende ni es adecuado.
- Con respecto a "modelado de objetos", el problema es que el término "objeto" es data- mente un sinónimo de "entidad" en este contexto, si consideramos que se emplea en otros contextos con un significado completamente diferente (en particular, en otros contextos de *bases de datos;* vea la parte VI de este libro). En realidad, nos parece que justo este hecho (que el término tenga dos significados) es responsable de lo que en otra parte de nominamos **el primer gran error garrafal** [3.3]. Vea el capítulo 25 para ahondarmá en este punto.

Regresemos al punto principal de la exposición. Nuestro motivo para incluir este material en esta parte del libro es el siguiente: *las ideas del modelado semántico pueden ser útiles com un auxiliar en el diseño de bases de datos, incluso en ausencia de un soporte directo del DBMS para dichas ideas.* De ahí que, tal como las ideas del modelo relacional original se usaron como un auxiliar en el diseño de bases de datos primitivas mucho antes de que hubiera alguna implementación comercial de ese modelo, así las ideas de algún modelo "extendido" podrían ser útiles como auxiliares en el diseño aunque no haya implementaciones comerciales de esas ideas. De hecho, al momento de la publicación de este libro, podríamos decir con certeza que el mayor **impacto** de las ideas del modelado semántico se ha dado en el área del diseño de bases de datos;* han propuesto varias metodologías de diseño basadas en uno u otro enfoque de modelado semántico. Por esta razón el énfasis principal de este capítulo está en la aplicación de las ideas del modelado semántico específicamente en el aspecto del diseño de bases de datos.

El plan del capítulo es el siguiente. Después de la introducción, la sección 13.2 explica en términos generales lo que involucra el modelado semántico. Posteriormente, la sección 13.3 presenta el más conocido de los modelos extendidos, el modelo *entidad/vínculo* (o E/R) de Chen, y las secciones 13.4 y 13.5 consideran la aplicación de ese modelo al diseño de bases de datos. (Explicamos brevemente otros modelos en las notas a algunas de las referencias de la sección "Referencias y bibliografía".) Por último, la sección 13.6 ofrece un breve análisis de ciertos aspectos del modelo E/R, y la sección 13.7 presenta un resumen.

13.2 EL ENFOQUE GENERAL

Podemos caracterizar el enfoque general para el problema del modelado semántico en términos de los cuatro pasos siguientes:

1. Primero, intentamos identificar un conjunto de conceptos *semánticos* que parezcan ser útiles al hablar informalmente acerca de la realidad. Por ejemplo:
   - Podríamos acordar que la realidad está conformada por **entidades.** (A pesar del hecho de que no podemos declarar con precisión qué es exactamente una entidad, el concepto de entidad parece ser útil para hablar de la realidad, por lo menos de manera intuitiva).
   - Podríamos ir aún más allá y acordar que las entidades pueden ser clasificadas de manera útil dentro de **tipos de entidad.** Por ejemplo, podríamos acordar que todos los emplea dos individuales son **ejemplares** del **tipo** de entidad genérico EMPLEADO. La ventaja de dicha clasificación es que todas las entidades de un tipo dado tendrán ciertas **pro piedades** en común (por ejemplo, todos los empleados tienen un salario) y por lo tanto que dicha clasificación puede conducir a cierto *ahorro de representaciones* (bastante ob vios). Por ejemplo, en términos relaciónales, estos aspectos comunes pueden convertirse en el encabezado de una varrel.
   - Podríamos ir aún más lejos y acordar que toda entidad tiene una propiedad especial que sirve para *identificar* a esa entidad; es decir, cada entidad tiene una **identidad.**
   - Podríamos ir de nuevo más lejos y acordar que cualquier entidad puede relacionarse con otras entidades por medio de **vínculos.**

   Y así sucesivamente. Aunque observe con detenimiento que todos estos términos (entidad, ejemplar, tipo de entidad, propiedad, vínculo, etcétera) *no* están precisa ni formalmente definidos; son conceptos "de la realidad", no formales. El paso 1 no es formal. En contraste, los pasos 2 a 4 siguientes son formales.
2. A continuación, tratamos de crear un conjunto de **objetos** *simbólicos* (es decir, formales) correspondientes, que puedan ser usados para representar los conceptos semánticos anteriores. *(Nota:* ¡ Aquí no estamos usando el término *objeto* con algún sentido intencionado!) Por ejemplo, el *modelo relacional extendido* RM/T [13.6] proporciona algunas clases especiales de relaciones, denominadas *relaciones E* y *P.* Por así decirlo, las relaciones E representan entidades y las relaciones P representan propiedades; sin embargo, las relaciones E y P tienen por supuesto definiciones formales, mientras que (como explicamos antes) las entidades y las propiedades no.

3. También creamos un conjunto general de **reglas de integridad** formales (o "metarrestric ciones", para emplear la terminología del capítulo 8) que vayan junto con esos objetos foi males. Por ejemplo, el RM/T incluye una regla llamada *integridad de propiedad,* la cual dice que toda entrada en una relación P debe tener una entrada correspondiente en una relación E (para reflejar el hecho de que toda propiedad en la base de datos debe ser una pn piedad de alguna entidad).
4. Por último, también desarrollamos un conjunto de **operadores** formales para manipular di chos objetos formales. Por ejemplo, el RM/T proporciona un operador *PROPERTY* que puede ser usado para juntar una relación E con todas sus relaciones P correspondiente —independientemente de cuántas haya o cuáles sean sus nombres—, lo que nos permi reunir todas las propiedades de una entidad cualquiera.

Los objetos, reglas y operadores de los pasos 2 al 4 anteriores, constituyen en su conjunto un modelo de datos extendido —es decir, "extendido" cuando esas construcciones son en verdad un superconjunto de aquellas correspondientes a uno de los modelos básicos, como el modelo relacional básico—, pero en este contexto en realidad no existe una distinción clara entre lo que es extendido y lo que es básico. Observe con detenimiento que *las reglas y operador son parte del modelo, como lo son los objetos* (por supuesto, tal como lo son en el modelo relacional básico). Por otra parte, quizá sea justo decir que los operadores son menos importantes que los objetos y las reglas, desde el punto de vista del diseño de bases de datos; por lo tanto, en el resto del capítulo, el énfasis está en los objetos y en las reglas en vez de los operadores (aunque en su momento ofreceremos algunos comentarios con respecto a los operadores).

Para repetir, el paso 1 intenta identificar un conjunto de conceptos semánticos que puedan ser útiles al hablar acerca de la realidad. La figura 13.1 muestra algunos de estos conceptos —entidad, propiedad, vínculo, subtipo—junto con definiciones informales y algunos ejemplos. Observe que elegimos deliberadamente los ejemplos para ilustrar la idea de que el mismo obje de la realidad podría ser legítimamente considerado por una persona como una entidad, por otras como una propiedad y por otras más como un vínculo. (Por cierto, esta idea muestra por que es imposible dar una definición precisa a términos como "entidad".) Soportar esta *flexibilidad ie interpretación* es una meta del modelado semántico (lo cual no ha sido logrado en su totalidad!.

Por cierto, observe que es probable que haya conflictos entre (a) términos como los que ilustra la figura 13.1 y que son utilizados en el nivel semántico y (b) términos utilizados en algúi formalismo subyacente como el del modelo relacional. Por ejemplo, muchos esquemas del mo delado semántico usan el término *atributo* en lugar del nuestro *propiedad,* pero no necesariamente indica que dicho atributo sea lo mismo que un atributo en el nivel relacional ni que se transforme en uno. Como otro ejemplo (importante), el concepto de *tipo de entidad* —como se usa en (po ejemplo) el modelo E/R— no es lo mismo que el concepto de *tipo* que explicamos en el capítulo 5. Para ser más específicos, es probable que dichos tipos de entidad se transformen *envarreh* de un diseño relacional; de modo que en realidad no corresponden a los tipos de *atributo* «*t* cionales (dominios). Pero tampoco corresponden por completo a los tipos de *relación,* debido a que:

1. Al nivel semántico, algunos tipos de relación base corresponderán probablemente a tipt de vínculos, no a tipos de entidad, y
2. Los tipos de relación derivados podrían no corresponder a nada en absoluto en el nivel semántico (hablando en términos generales).

Concepto	Definición informal	Ejemplos
ENTIDAD	Un objeto distinguible	Proveedor, parte, envío Empleado, departamento Persona Composición, concierto Orquesta, director Orden de compra, Línea de pedido
PROPIEDAD	Una pieza de información que describe una entidad	Número de proveedor Cantidad del envío Departamento del empleado Altura de la persona Tipo de concierto Fecha de orden de compra
VINCULO	Una entidad que sirve para interconectar dos o más entidades	Envío (proveedor-parte) Asignación (empleado-departamento) Grabación (composición-orquesta-director)
SUBTIPO	El tipo de entidad Y es un subtipo del tipo de entidad X si y sólo si toda Y es necesariamente una X	Empleado es un subtipo de Persona Concierto es un subtipo de Composición

Figura 13.1 Algunos conceptos semánticos útiles.

La confusión sobre los niveles —en particular, la confusión que surge de dichos conflictos de terminología— ha conducido en el pasado a algunos errores costosos, y sigue sucediendo hasta la fecha (vea el capítulo 25, sección 25.2).

Una última observación para cerrar esta sección. En el capítulo 1 señalamos que los vínculos se conciben mejor como entidades por derecho propio y que en este libro los trataríamos generalmente de esa forma. También señalamos en el capítulo 3 que una ventaja del modelo relacional era precisamente que representaba a todas las entidades (incluyendo a los vínculos) de la misma manera uniforme; es decir, por medio de varrels. Sin embargo, el concepto de vínculo (como el concepto de entidad) parece ser *intuitivamente* útil al hablar acerca de la realidad; es más, el enfoque para el diseño de bases de datos que explicaremos en las secciones 13.3 a 13.5, depende en gran medida de la distinción de entidad frente a vínculo. Por lo tanto, adoptamos la terminología de vínculo para los propósitos de las secciones que siguen. Sin embargo, tendremos más que decir al respecto en la sección 13.6.

13.3 EL MODELO E/R

Como señalamos en la sección **13.1,** uno de los enfoques más conocidos del modelado semántico —en realidad, uno de los más ampliamente utilizados— es el tan mencionado enfoque **entidad/vínculo** (o E/R), basado en el "modelo entidad/vínculo" que introdujo Chen en 1976 [13.5], y que desde entonces el mismo Chen y otros más refinaron en diversas formas (vea por ejemplo las referencias [13.13] y [13.40-13.42]). Por lo tanto, la mayor parte de este capítulo está dedicada a una explicación del enfoque E/R. (Sin embargo, debemos subrayar que el modelo E/R está muy lejos de ser el único modelo "extendido"; se han propuesto muchos otros. Para conocer otros más, vea por ejemplo las referencias [13.5], [13.13], [13.25] y en particular la referencia [13.19], así como las referencias [13.22] y [13.31] para investigaciones tutoriales en el campo.)

El modelo E/R incluye equivalentes de todos los objetos semánticos que mostramos en la figura 13.1. Los examinaremos uno por uno. Sin embargo, antes debemos señalar que la referencia [13.5] no sólo presentó el modelo E/R como tal, también presentó una **técnica de elaboración de diagramas** correspondiente (los "diagramas E/R"). En la siguiente sección explicaremos con cierto detalle los diagramas E/R, aunque la figura 13.2 muestra un ejemplo sencillo de dichos diagramas (basado en una figura de la referencia [13.5]), podría encontrarlo útil para estudiar este ejemplo junto con las explicaciones de la presente sección. El ejemplo representa los datos de una sencilla compañía manufacturera (es una versión ampliada del diagrama E/R que dimos para la compañía "KnowWare Inc." en la figura 1.6 del capítulo 1).

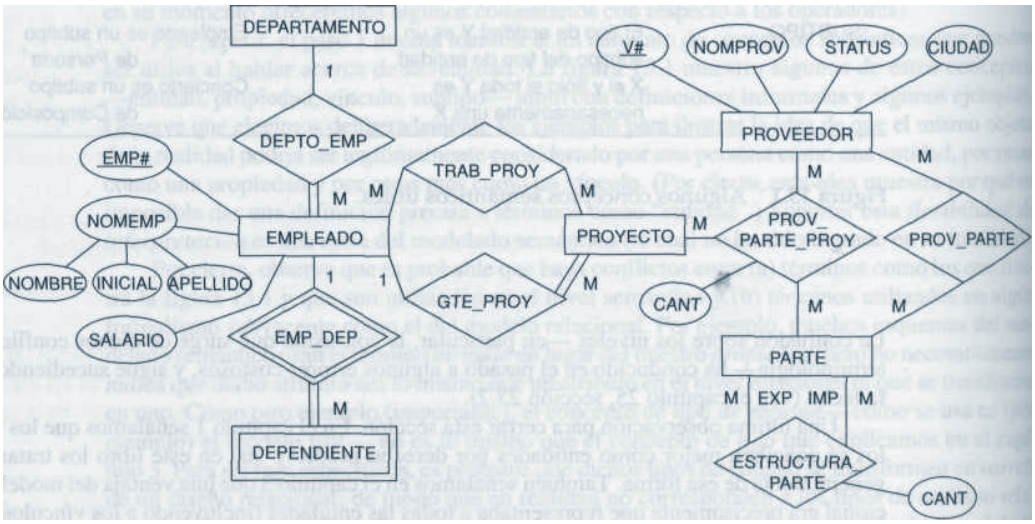


Figura **13.2** Diagrama entidad/vínculo (ejemplo incompleto).

Nota: La mayoría de las ideas que explicaremos en las siguientes subsecciones, serán bastante familiares para cualquiera que conozca el modelo relacional. Sin embargo (como verá). existen ciertas diferencias en la terminología.

Entidades

La referencia [13.5] comienza por definir una **entidad** como "algo que puede ser identificado en forma distintiva". Después continúa con la clasificación de las entidades en **entidades normales y entidades débiles.** Una entidad débil es aquella cuya existencia depende de alguna otra entidad, en el sentido de que no puede existir si esa otra entidad no existe también. Por ejemplo, con referencia a la figura 13.2, las personas a cargo de un empleado podrían ser entidades débiles; no pueden existir (por lo que a la base de datos concierne) si no existe el empleado relevante. En particular, si se elimina un empleado determinado, es necesario eliminar también a todas las personas que están a su cargo. En contraste, una entidad normal es aquella que no es débil; por ejemplo, los empleados podrían ser entidades normales. *Nota:* Algunos autores usan el término "entidad fuerte" en lugar de "entidad normal".

Propiedades

Las entidades —y también los vínculos— tienen **propiedades.** Todas las entidades o vínculos de un tipo determinado tienen ciertas clases de propiedades en común. Por ejemplo, todos los empleados tienen un número de empleado, un nombre, un salario, etcétera. *(Nota:* Con toda intención, no mencionamos aquí el "número de departamento" como una propiedad del empleado. Vea más adelante la explicación sobre vínculos.) Cada clase de propiedad obtiene sus valores de un **conjunto de valores** correspondiente (es decir, de un dominio, en términos relaciónales). Además, una propiedad puede ser:

- **Simple** o **compuesta.** Por ejemplo, la propiedad compuesta "nombre del empleado" podría estar conformada por las propiedades simples "nombre", "inicial media" y "apellido".
- **Clave** (es decir, única, posiblemente dentro de algún contexto). Por ejemplo el nombre de un dependiente podría ser único dentro del contexto de un empleado dado.
- **Monovaluada** o **multivaluada** (en otras palabras, se permiten grupos repetitivos). Todas las propiedades que muestra la figura 13.2 son monovaluadas, pero si (por ejemplo) un proveedor dado pudiera tener varias ubicaciones distintas, entonces "ciudad del proveedor" podría ser una propiedad multivaluada.
- **Faltante** (por ejemplo, "desconocida" o "no aplicable"). Este concepto no está ilustrado en la figura 13.2. Para una exposición detallada, consulte el capítulo 18.
- **Base** o **derivada.** Por ejemplo, la "cantidad total" de una parte en particular podría ser derivada como la suma de las cantidades de los envíos individuales de esa parte. Una vez más, este concepto no está ilustrado en la figura 13.2.

Nota: Algunos autores emplean el término "atributo" en lugar de "propiedad" en un contexto de E/R.

Vínculos

La referencia [13.5] define un **vínculo** como "una asociación entre entidades". Por ejemplo, existe un vínculo llamado DEPTO_EMP entre departamentos y empleados, que representa el hecho de que cierto departamento emplea a ciertos empleados. Al igual que con las entidades

(vea el capítulo 1), es necesario distinguir en principio entre los tipos de vínculo y los ejemplar de vínculo; aunque en una explicación informal es común ignorar tales refinamientos, y nosotro lo haremos constantemente a partir de este punto.

Decimos que las entidades involucradas en un vínculo dado son **participantes** en ese vínculo. Al número de participantes en un vínculo dado se le llama **grado** de ese vínculo. (Por lo tamo. observe que este término no significa lo mismo que en el modelo relacional.)

Sea *R* un tipo de vínculo que involucra como participante al tipo de entidad *E.* Si todo ejemplar de *E* participa en por lo menos un ejemplar de *R,* entonces decimos que la participación de *E* en *R* es **total;** en caso contrario, decimos que es **parcial.** Por ejemplo, si todo empleado debe pertenecer a un departamento, entonces la participación de empleados en DEPTO_EMP es **tot** si es posible que un departamento dado no tenga ningún empleado, entonces la participación d¡ departamentos en DEPTO_EMP es parcial.

Un vínculo E/R puede ser **uno a uno, uno a muchos** (también conocido como muchos a **uno)** o **muchos a muchos** (para simplificar damos por hecho que todos los vínculos son bir ríos; es decir, de grado dos. Por supuesto, ampliar los conceptos y la terminología a vínculos d mayor grado es algo directo). Ahora bien, si está usted familiarizado con el modelo relacional, podría verse tentado a pensar en el caso muchos a muchos como en el único que es un vínculo genuino, ya que ese caso es el único que demanda una representación por medio de una varrel independiente; los vínculos uno a uno y uno a muchos siempre pueden ser representados por medio de una clave externa en una de las varrels participantes. Sin embargo, existen buenas razones para tratar a los casos uno a uno y uno a muchos del mismo modo que el caso de muchos a muchos, por lo menos si existe cualquier posibilidad de que con el paso del tiempo puedan evolucionar y convertirse en muchos a muchos. Sólo si no existe dicha posibilidad es seguro tratarlos de manera diferente. Desde luego, en ocasiones *no* existe esa posibilidad; por ejemplo, siempre será cierto que un círculo tiene exactamente un punto en su centro.

Subtipos y supertipos de entidades

Nota: Las ideas que expusimos en esta subsección no fueron incluidas en el modelo E/R original [13.5], pero fueron incorporadas después. Por ejemplo, vea Teorey, Yang y Fry [1S.41].

Cualquier entidad dada tiene por lo menos un tipo de entidad, aunque una entidad puede ser de varios tipos al mismo tiempo. Por ejemplo, si algunos empleados son programadores (y todos los programadores son empleados), entonces podríamos decir que el tipo de entidad PROGRAMADOR es un **subtipo** del tipo de entidad EMPLEADO (o de manera equivalente, que el tipo de entidad EMPLEADO es un **supertipo** del tipo de entidad PROGRAMADOR). Todas las propiedades de los empleados se aplican automáticamente a los programadores, pero no sucede lo contrario (por ejemplo, los programadores podrían tener una propiedad "aptitud en lenguaje de programación" que no es aplicada a los empleados en general). En forma similar, los progn madores participan automáticamente en todos los vínculos en los que participan los empleados, pero lo contrario no es cierto (por ejemplo, los programadores podrían pertenecer a alguna sociedad de computación profesional, mientras que los empleados en general no). Decimos que las propiedades y vínculos que se aplican al supertipo son **heredados** por el subtipo.

Observe además que algunos programadores podrían ser programadores de aplicaciones y otros programadores de sistemas; por lo que podríamos decir que los tipos de entidad PROGRAMADOR_APLICACIONES y PROGRAMADOR_SISTEMAS son subtipos del supertipo PROGRAMADOR (y así sucesivamente). En otras palabras, un subtipo de entidad sigue

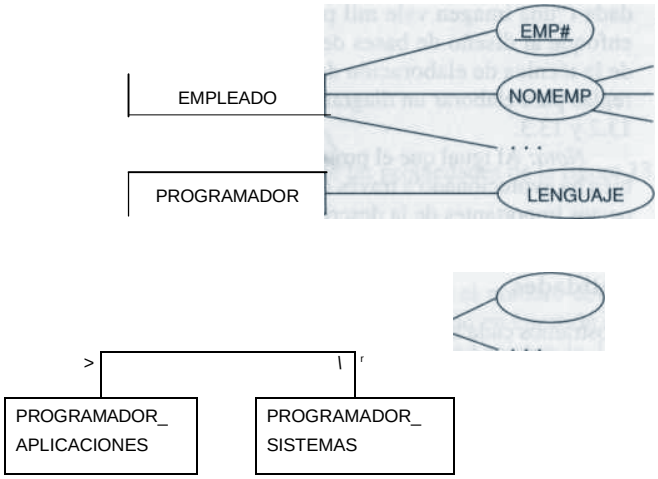


Figura 13.3 Ejemplo de una jerarquía de tipos de entidad.

siendo un tipo de entidad y por lo tanto puede tener sus propios subtipos. Un tipo de entidad dado, sus subtipos inmediatos, los subtipos inmediatos de éstos y así sucesivamente, constituyen juntos una **jerarquía de tipos de entidad** (vea la figura 13.3). Puntos a destacar:

1. En el capítulo 19 explicaremos a profundidad las jerarquías de tipos y la herencia de tipos. Sin embargo, ya desde ahora debemos advertirle que en ese capítulo usaremos el término *tipo* para referirnos exactamente al significado que dimos en el capítulo 5; *no* significará "tipo de entidad" en el sentido del presente capítulo.
2. Si usted está familiarizado con IMS (o con algún otro sistema de base de datos que soporte una estructura jerárquica de datos), debe notar que las jerarquías de tipos *no* son como las jerarquías al estilo de IMS. Por ejemplo, en la figura 13.3 no hay ninguna sugerencia de que para un EMPLEADO haya varios PROGRAMADORes correspondientes (como sucedería si la figura representase una jerarquía al estilo de IMS); por el contrario, para un ejemplar de EMPLEADO existe *como máximo un* PROGRAMADOR correspondiente que repre senta al mismo EMPLEADO en su papel de PROGRAMADOR.

Esto nos lleva al final de nuestra breve explicación sobre las principales características estructurales del modelo E/R. Enfoquemos ahora nuestra atención a los diagramas E/R.

13.4 DIAGRAMAS E/R

Como explicamos en la sección anterior, la referencia [13.5] no sólo presentó el modelo E/R como tal, sino que también presentó el concepto de **diagramas entidad/vínculo** (o diagramas E/R). Los diagramas E/R constituyen una técnica para representar la estructura lógica de una base de datos en forma de gráficos. Como tales, proporcionan un medio sencillo y de fácil comprensión para comunicar las características sobresalientes del diseño de cualquier base de datos

dada ("una imagen vale mil palabras"). De hecho, la popularidad del modelo E/R como un enfoque al diseño de bases de datos puede ser atribuido más probablemente a la existenci de la técnica de elaboración de diagramas E/R que a cualquier otra causa. Describiremos la reglas para elaborar un diagrama E/R en términos de los ejemplos que ya dimos en las figuras 13.2 y 13.3.

Nota: Al igual que el propio modelo E/R, la técnica de elaboración de diagramas E/R tan bien ha evolucionado a través del tiempo. La versión que describimos aquí difiere en ciertos ¡ pectos importantes de la descrita originalmente en la referencia [13.5].

Entidades

Mostramos cada tipo de entidad como un rectángulo que contiene el nombre del tipo de entidad en cuestión. Para las entidades débiles, el borde del rectángulo es doble.

Ejemplos (vea la figura 13.2):

- Entidades normales:

  DEPARTAMENTO
  EMPLEADO
  PROVEEDOR
  PARTE
  PROYECTO
- Entidad débil:
  DEPENDIENTE

Propiedades

Mostramos las propiedades como elipses que contienen el nombre de la propiedad en cuestiói y que están conectadas a la entidad o vínculo relevante mediante una línea continua. El borde de la elipse es punteado cuando la propiedad es derivada y doble cuando es multivaluada. Sil propiedad es compuesta, sus propiedades componentes son mostrados en elipses adicionales que están conectadas a la elipse de la propiedad compuesta en cuestión por medio de lincas conti nuas. Las propiedades que son claves aparecen subrayadas. No mostramos los conjuntos de valores correspondientes a las propiedades.

Ejemplos (vea la figura 13.2):

- Para EMPLEADO:

  EMP# (clave)
  NOMEMP (compuesta, consistente de NOMBRE, INICIAL y APELLIDO)
  SALARIO
- Para PROVEEDOR:

  V# (clave)
  NOMPROV
  STATUS
  CIUDAD

Para PR0V_PARTE_PROY:

CANT

Para ESTRUCTURA_PARTE:

CANT

Por razones de espacio, omitimos el resto de las propiedades de la figura 13.2.

Vínculos

Mostramos cada tipo de vínculo en un rombo que contiene el nombre del tipo de vínculo en cuestión. El borde del rombo aparece doble cuando el vínculo en cuestión es el que está entre un tipo de entidad débil y el tipo de entidad del cual depende su existencia. Los participantes en cada vínculo están conectados por medio de líneas continuas; cada una de estas líneas tiene una etiqueta "1" o "M" para indicar cuando el vínculo es uno a uno, muchos a uno, etcétera. Si la participación es total, la línea es doble.

Ejemplos (vea la figura 13.2):

- DEPTO_EMP (vínculo uno a muchos entre DEPARTAMENTO y EMPLEADO).
- EMP_DEP (vínculo uno a muchos entre EMPLEADO y DEPENDIENTE, un tipo de en tidad débil).
- TRAB_PROY y GTE_PROY (vínculo muchos a muchos y uno a muchos, respectivamente, entre EMPLEADO y PROYECTO).
- PROV_PARTE_PROY (vínculo muchos a muchos que involucra a PROVEEDOR, PARTE y PROYECTO).
- PROV_PARTE (vínculo muchos a muchos entre PROVEEDOR y PARTE).
- ESTRUCTURA_PARTE (vínculo muchos a muchos entre PARTE y PARTE).

Observe que en el último ejemplo las dos líneas desde PARTE hacia ESTRUCTURA_PARTE se distinguen etiquetándolas con dos nombres distintos, de acuerdo con el **papel** que desempeñan (EXP e IMP, para "explosión de partes" e "implosión de partes", respectivamente). ESTRUCTURA_PARTE es un ejemplo de lo que en ocasiones se denomina **vínculo recursivo.**

Subtipos y supertipos de entidades

Sea el tipo de entidad *Y* un subtipo del tipo de entidad *X*. Entonces dibujamos una línea continua desde el rectángulo *X* hacia el rectángulo *Y*, con una flecha en el extremo *Y*. La línea indica lo que a veces se llama "el vínculo *es una"* (debido a que toda *Y* "es una" *X;* en forma similar, el conjunto de todas las Ks es un subconjunto del conjunto de todas las *Xs).*

Ejemplos (vea la figura 13.3):

- PROGRAMADOR es un subtipo de EMPLEADO.
- PR0GRAMADOR_APLICACIONES y PROGRAMADOR_SISTEMAS son subtipos de PROGRAMADOR.

13.5 DISEÑO DE BASES DE DATOS CON EL MODELO E/R

En cierto sentido, un diagrama E/R construido de acuerdo con las reglas esbozadas en la anterior *es* un diseño de base de datos. Sin embargo, si intentamos transformar un diseño así dentro de los formalismos de un DBMS específico,* pronto descubriremos que el diagrama E/R es aún muy impreciso en ciertos aspectos y deja sin especificar varios detalles (en especial, **detall*** de restricciones de integridad). Para ilustrar la idea, consideremos lo que involucra transforma el diseño de la figura 13.2 en una definición de base de datos relacional.

Entidades normales

Para repetir, las entidades normales de la figura 13.2 son las siguientes:

DEPARTAMENTO
EMPLEADO
PROVEEDOR
PARTE
PROYECTO

Cada tipo de entidad normal se transforma en una varrel base. De esta manera, la base de datos contendrá cinco varrels base —digamos DEPTO, EMP, V, P y Y— correspondientes a estos cinco tipos de entidad. Además, cada una de esas cinco varrels base tendrán una clave candidata —representadas, digamos, por los atributos DEPTO#, EMP#, V#, P# y Y#— correspondiente para las "claves" identificadas en el diagrama E/R. Por razones de seguridad, acordemos dar específicamente a cada varrel una clave *primaria.* Entonces la definición de (por ejemplo) la varrel DEPTO podría comenzar viéndose de la siguiente forma:

```
VAR DEPTO BASE RELATION
        { DEPT0# ..., ... } PRIMARY
        KEY { DEPTO# } ;
```

Dejamos como ejercicio las otras cuatro varrels. *Nota:* Por supuesto, también es necesario definir los dominios o "conjuntos de valores". Omitimos aquí la explicación detallada de este aspecto, debido a que (como ya mencionamos) el diagrama E/R no incluye los conjuntos de valores.

Vínculos muchos a muchos

Los vínculos muchos a muchos (o muchos a muchos a muchos, etcétera) del ejemplo son los siguientes:

TRAB_PROY (que involucra a empleados y proyectos). PROV_PARTE (que involucra a proveedores y partes). PROV_PARTE_PROY (que involucra a proveedores, partes y proyectos). ESTRUCTURA_PARTE (que involucra a partes y partes).

---

*Ahora existen muchas herramientas que pueden ayudar en ese proceso de transformación (por ejemplo, mediante la utilización del diagrama E/R para generar instrucciones CREATE TABLE de SQL y similares).

Cada uno de estos vínculos se transforma también en una varrel base. Por lo tanto presentamos cuatro varrels base adicionales, correspondientes a estos cuatro vínculos. Concentrémonos en el vínculo PROV_PARTE. Sea VP la varrel para ese vínculo (la varrel usual de envíos). Permitámonos diferir por un momento la cuestión de la clave primaria de esta varrel y concentrémonos en su lugar en el asunto de las claves *externas* que son necesarias para identificar a los participantes en el vínculo:

```
VAR VP BASE RELATION
    { V# ...,  P# ...,... }

    FOREIGN KEY { V# } REFERENCES V
    FOREIGN KEY { P# } REFERENCES P ;
```

Resulta claro que la varrel debe incluir dos claves externas (V# y P#) correspondientes a los dos participantes (proveedores y partes), y dichas claves externas deben hacer referencia a las varrels participantes correspondientes (V y P). Además, debemos especificar un conjunto adecuado de *reglas de clave externa* —es decir, una regla DELETE y una regla UPDATE— para cada una de dichas claves externas (si necesita refrescar su memoria con respecto a estas reglas, consulte el capítulo 8). En el caso de la varrel VP, podríamos especificar las reglas como sigue. (Por supuesto, las reglas específicas que mostramos son sólo a manera de ejemplo; observe en particular que *no* son derivables de ni especificadas por el diagrama E/R.)

```
VAR VP BASE RELATION
    { V# ...,  P# ...,... }

    FOREIGN KEY { V# } REFERENCES V
            ON DELETE RESTRICT
            ON UPDATE CASCADE FOREIGN
    KEY { P# } REFERENCES P
            ON DELETE RESTRICT
            ON UPDATE CASCADE ;
```

¿Qué hay con respecto a la clave primaria de esta varrel? Una posibilidad sería tomar la combinación de las claves externas que identifican a los participantes (V# y P#, en el caso de VP), *cuando* (a) esa combinación tiene un valor único para cada ejemplar del vínculo (lo cual podría o no ser el caso, aunque por lo regular lo es) y *cuando* (b) el diseñador no tiene objeción en cuanto a las claves primarias compuestas (lo cual podría o no ser el caso). Como alternativa, podríamos presentar un nuevo atributo *sustituto* no compuesto —digamos "número de envío"— que sirva como clave primaria (vea las referencias [13.10] y [13.16]). Para efectos del ejemplo, abordaremos la primera de estas dos posibilidades y agregaremos así la cláusula

```
PRIMARY KEY { V#, P# }
```

a la definición de la varrel base VP.

Dejamos como ejercicio la consideración de los vínculos TRAB_PROY, ESTRUCTURA_PARTE y PROV_PARTE_PROY.

Vínculos muchos a uno

En el ejemplo hay tres vínculos muchos a uno:

GTE_PROY (de proyectos a gerentes).

DEPTO_EMP (de empleados a departamentos).

EMP_DEP (de dependientes a empleados).

De estos tres, el último involucra un tipo de entidad débil (DEPENDIENTE); los otros do; sólo involucran tipos de entidad normales. En un momento más explicaremos el caso de la tidad débil, pero por ahora, concentrémonos en los otros dos casos. Considere el ejer DEPTO_EMP. Este ejemplo *no* ocasiona la introducción de ninguna varrel nueva.* En su lugar introducimos simplemente una clave externa en la varrel (EMP) que está del lado "muchos" vínculo, el cual hace referencia a la varrel (DEPTO) que está del lado "uno"; de esta manera i nemos:

```
VAR EMP BASE RELATION
     { EMP# ..., DEPT0# ... , ... }
     PRIMARY KEY { EMP# }
     FOREIGN KEY { DEPTO* } REFERENCES DEPTO
             ON DELETE . ..
             ON UPDATE ... ;
```

Aquí, las posibilidades de las reglas DELETE y UPDATE son (en general) exactamente las mismas que para una clave externa que representa a un participante en un vínculo muchos a muchos. Observe una vez más que no están especificadas en el diagrama E/R.

Nota: Para efectos de la presente explicación, damos por hecho que los vínculos uno a uno (que en todo caso no son tan comunes en la práctica) se tratan exactamente de la misma manera que los vínculos muchos a uno. La referencia [13.7] contiene una explicación más amplia de problema especial del caso uno a uno.

Entidades débiles

El vínculo de un tipo de entidad débil con el tipo de entidad del cual depende, es por supuesto un vínculo muchos a uno (como indicamos en la subsección previa). Sin embargo, las regla DELETE y UPDATE para ese vínculo *deben* ser como sigue:

```
ON DELETE CASCADE
ON UPDATE CASCADE
```

Estas especificaciones en conjunto, captan y reflejan la existencia necesaria de dependencia. Aquí tenemos un ejemplo:

```
VAR DEPENDIENTE BASE RELATION {
     EMP# ... , ... }

     FOREIGN KEY { EMP# } REFERENCES EMP
             ON DELETE CASCADE ON UPDATE
             CASCADE ;
```

¿Cuál es la clave primaria de una varrel como ésta? Como en el caso de los vínculos mu chos a muchos, resulta que tenemos opciones a elegir. Una posibilidad es tomar la combinación de la clave externa y la "clave" de la entidad débil del diagrama E/R, *si* (una vez más) el diseña- dor de la base de datos no tiene objeción en cuanto a las claves primarias compuestas. Como al- ternativa, podríamos introducir un nuevo atributo sustituto (no compuesto) que sirva como clave

*Aunque podría hacerlo; como mencionamos en la sección 13.3, a veces existen buenas razones para tratar a un vínculo muchos a uno como si en realidad fuera muchos a muchos. Para una mayor explicación, vea por ejemplo la parte final de la referencia [18.20].

primaria (de nueva cuenta, vea las referencias [13.10] y [13.16]). Para efectos del ejemplo abordaremos otra vez la primera de las dos posibilidades y así, agregamos la cláusula

```
PRIMARY KEY { EMP#, N0MJ5EP }
```

(donde N0M_DEP es el nombre del dependiente del empleado) a la definición de la varrel base DEPENDIENTE.

Propiedades

Cada propiedad mostrada en el diagrama E/R se transforma en un atributo de la varrel apropiada; salvo que si la propiedad está multivaluada, crearíamos normalmente una nueva varrel para ella de acuerdo con los principios de la normalización que explicamos en el capítulo 11 (en especial en la sección 11.6). Se crean —de la manera obvia— dominios para los conjuntos de valores (aunque, en primer lugar, la decisión de los conjuntos de valores podría no ser tan obvia). Los detalles de estos pasos son directos y los omitimos aquí.

Subtipos y supertipos de entidades

Puesto que la figura 13.2 no contiene ningún ejemplo de subtipos y supertipos, pasemos al ejemplo de la figura 13.3 que sí los tiene. Concentrémonos por el momento en los tipos de entidad EMPLEADO y PROGRAMADOR. Para simplificar, suponga que los programadores sólo manejan un lenguaje de programación (es decir, la propiedad LENGUAJE es monovaluada). Entonces:*

- El supertipo EMPLEADO se transforma en una varrel base (digamos EMP) de la manera usual que ya explicamos.
- El subtipo PROGRAMADOR se transforma en otra varrel base (digamos PGMR) con la misma clave primaria que la varrel de supertipo y con otros atributos correspondientes a las propiedades que se aplican sólo a los empleados que son programadores (en el ejemplo, so lamente LENGUAJE):

  ```
  VAR PGMR BASE RELATION
       { EMP# ..., LENGUAJE ... }
       PRIMARY KEY { EMP# } . . . ;
  ```

  Además, la clave primaria de PGMR es también una clave *externa,* que hace referencia de nuevo a la varrel EMP. Por lo tanto, necesitamos ampliar la definición de acuerdo con esto (observe en particular las reglas DELETE y UPDATE):

  ```
  VAR PGMR BASE RELATION
       { EMP# ..., LENGUAJE ... }
       PRIMARY KEY { EMP# }
       FOREIGN KEY { EMP# } REFERENCES EMP
               ON DELETE CASCADE
               ON UPDATE CASCADE ;
  ```

"Observe en particular que lo que *no* haremos posteriormente es transformar a los empleados y a los programadores en alguna clase de construcciones de "supertabla" y "subtabla". Aquí existe una dificultad conceptual, o por lo menos una trampa: sólo porque el tipo de entidad *Y* es un subtipo del tipo de entidad *X* en el diagrama E/R, no podemos deducir que el equivalente relacional de *Y* sea un "sub" *algo* del análogo relacional de *X,* y de hecho no lo es. Para una mayor explicación, vea la referencia [13.12].

- También necesitamos una *vista* (digamos EMP_PGMR) que sea la junta de las varrels supertipo y subtipo:

```
VAR EMPPGMR VIEW EMP
    JOIN PGMR ;
```

Observe que esta junta es (cero o uno) a uno; está sobre una clave candidata y una clave externa coincidente, y la clave externa es en sí misma una clave candidata. Entonces, en térmim generales, la vista contiene sólo aquellos empleados que son programadores.

Dado este diseño:

- Podemos acceder a las propiedades que se aplican a todos los empleados (por ejemplo, para fines de recuperación) mediante el uso de la varrel base EMP.
- Podemos acceder a las propiedades que se aplican sólo a programadores mediante la varrel base PGMR.
- Podemos acceder a *todas* las propiedades que se aplican a programadores por medio de la vista EMP_PGMR.
- Podemos insertar empleados que no son programadores utilizando la varrel base EMP.
- Podemos insertar empleados que son programadores por medio de la vista EMPPGMR.
- Podemos eliminar empleados, programadores u otros, mediante la varrel EMP o (sólo a pro gramadores) mediante la vista EMP_PGMR.
- Podemos actualizar propiedades que se aplican a todos los empleados utilizando la varrel base EMP o (sólo a programadores) usando la vista EMP_PGMR.
- Podemos actualizar propiedades que se aplican sólo a programadores mediante la varrel PGMR.
- Podemos convertir en programador a un no programador existente insertando al empleado ya sea en la varrel base PGMR o en la vista EMP_PGMR.
- Podemos convertir en no programador a un programador existente eliminando a éste de la varrel base PGMR.

Dejamos como ejercicio la consideración de los otros tipos de entidades de la figura 13.3 (PROGRAMADOR.APLICACIONES y PROGRAMADOR_SISTEMAS).

13.6 UN BREVE ANÁLISIS

En esta sección examinamos brevemente, aunque con un poco de más profundidad, cienos aspectos del modelo E/R. Las explicaciones que siguen están tomadas en parte de un examen más amplio que realicé sobre los mismos temas (vea la referencia [13.8]). Usted puede encontrar análisis y comentarios adicionales en las notas de varias referencias que aparecen en la sección "Referencias y bibliografía" al final del capítulo.

¿El modelo E/R como fundamento del modelo relacional?

Comenzamos por considerar al enfoque E/R desde una perspectiva ligeramente diferente. Probablemente, para usted sea obvio que las ideas del enfoque E/R, o algo muy parecido a estas ideas, deben haber sido los antecedentes *informales* en la mente de Codd cuando desarrolló por primera

vez el modelo relacional *formal*. Como explicamos en la sección 13.2, el enfoque general para desarrollar un modelo "extendido" comprende cuatro pasos básicos, como sigue:

1. Identificar conceptos semánticos útiles.
2. Crear objetos formales.
3. Crear reglas de integridad formales ("metarrestricciones").
4. Crear operadores formales.

Pero estos mismos cuatro pasos pueden aplicarse también al diseño del modelo relacional *básico* (y de hecho a cualquier modelo de datos formal), no solamente a modelos "extendidos" como el E/R. En otras palabras, para que Codd construyera en primer lugar el modelo relacional básico (formal), debió haber tenido en mente algunos "conceptos semánticos útiles" (informales), y dichos conceptos deben haber sido básicamente los del modelo E/R (o algo muy parecido a ellos). De hecho, los propios escritos de Codd apoyan este punto de vista. En su primer artículo sobre el modelo relacional (la primera versión de la referencia [5.1]), encontramos lo siguiente:

> El conjunto de entidades de un tipo de entidad dado puede ser visto como una relación, y a dicha relación la llamaremos *relación de tipo de entidad*... Las relaciones restantes... están entre los tipos de entidad y se les... denomina *relaciones interentidad*... Una propiedad esencial de toda relación interentidad es que [incluye por lo menos dos claves externas que] se refieren ya sea a distintos tipos de entidad, o bien se refieren a un tipo de entidad común que tiene distintos papeles.

Aquí, Codd está proponiendo claramente que las relaciones sean usadas para representar tanto "entidades" como "vínculos". Pero —y éste es un pero muy grande— la idea es que *las relaciones son objetos formales y el modelo relacional es un sistema formal*. La esencia de la contribución de Codd fue que encontró un buen modelo *formal* para ciertos aspectos de la realidad.

En contraste con lo anterior, el modelo entidad/vínculo *no* es (o por lo menos, no de manera importante) un modelo formal. En vez de ello, consiste principalmente en un conjunto de conceptos informales, correspondientes (solamente) al paso 1 de los cuatro mencionados anteriormente. (Además, los aspectos formales que sí posee no parecen ser muy diferentes de los correspondientes en el modelo relacional básico; vea una explicación adicional de esta idea en la siguiente subsección.) Y aunque sin duda resulta útil contar con un arsenal de conceptos del "paso 1" para fines (entre otros) del diseño de bases de datos, se mantiene el hecho de que esos diseños de bases de datos no pueden ser completados sin los objetos formales y las reglas de los pasos 2 y 3; y no es posible llevar a cabo muchas otras tareas sin los operadores formales del paso 4.

Observe que las siguientes observaciones no pretenden sugerir que el modelo E/R no es útil. Lo es; pero eso no es todo. Además, resulta un poco extraño darse cuenta de que la primera descripción publicada del modelo E/R ¿«formal apareció varios años después de la primera descripción publicada del modelo relacional *formal*, dado que (como hemos visto) el último se basó originalmente en ideas más bien del tipo E/R.

¿Es el modelo E/R un modelo de datos?

De acuerdo con la explicación anterior, ni siquiera está claro si el "modelo" E/R es un verdadero modelo de datos del todo, al menos en el sentido en que hemos venido usando ese término hasta ahora (es decir, como un sistema formal que involucra aspectos estructurales, de integridad y de

manipulación). En realidad, por lo regular se usa el término "modelado E/R" para refei proceso de decidir (solamente) la *estructura* de la base de datos; aunque en nuestras explicaciones de las secciones 13.3 a 13.5,* también consideramos ciertos aspectos de integridad (que en su mayoría tienen que ver con las claves primaria y externa). Sin embargo, una lectura tolerante de la referencia [13.5] sugeriría que el modelo E/R es de hecho un modelo de dato aunque uno que en esencia es sólo *una capa delgada en la parte superior del modelo relación básico* (en realidad, no es un candidato a reemplazar el modelo relacional, como algunas personas han sugerido). Justificamos esta afirmación de la siguiente forma:

- Primero, el objeto de datos E/R fundamental —es decir, el objeto *formal* fundamental. I diferencia de los objetos informales "entidad", "vínculo", etcétera— es la relación naria.
- Los operadores E/R son básicamente los operadores del álgebra relacional. (En realidad, la referencia [13.5] no es muy clara en este punto, pero parece proponer un conjunto de opera dores que son estrictamente menos poderosos que los del álgebra relacional; por ejemplo. parece ser que no hay unión ni una junta explícita.)
- Es en el área de la integridad donde el modelo E/R tiene cierta funcionalidad (menor) que no posee el modelo relacional. El modelo E/R incluye un conjunto *integrado* de reglas de integridad que corresponden a algunas pero no a todas las reglas de clave externa explicadas en el presente libro. De ahí que, mientras un sistema relacional "puro" necesitaría que el usuario formulara de manera explícita ciertas reglas de clave externa, un sistema E/R sol requeriría que declarase que una varrel dada representa a cierta clase de vínculo, lo que en tonces haría más claro ciertas reglas de clave externa.

Entidades *vs.* vínculos

Ya hemos señalado varias veces en el libro que los "vínculos" se entienden mejor si los vemo: simplemente como un tipo especial de entidad. En contraste, es un *requisito* del enfoque E/R quf estos dos conceptos se distingan de alguna manera. En mi opinión, cualquier enfoque que insista en hacer tal distinción tiene serios defectos, debido a que (como mencionamos en la sección 13.2) *exactamente el mismo objeto* puede ser visto en forma legítima como una entidad por algunos usuarios y como un vínculo por otros. Considere por ejemplo el caso de un matrimonie

- Desde una perspectiva, el matrimonio es claramente un vínculo entre dos personas (consulta de ejemplo: "¿Con quién se casó Elizabeth Taylor en 1975?").
- Desde otra perspectiva, un matrimonio es claramente una entidad por derecho propio (con sulta de ejemplo: "¿Cuántos matrimonios se han realizado en esta iglesia desde el mes de abril?).

---

*Por supuesto, aquí existe una debilidad mayor: el modelo E/R es *completamente incapaz,* de tratar con re: tricciones de integridad o "reglas de negocios", con excepción de algunos casos especiales (que reconocemos como importantes). Aquí tenemos una cita típica: "las reglas declarativas son demasiado complejas para ser captadas como parte del modelo del negocio y deben ser definidas por separado por parte del analista o desarrollador" [13.27]. Y sin embargo, existe un fuerte argumento de que el diseño de bases de dalos debe ser —precisamente— un proceso de identificación de las restricciones aplicables (vea las referencias [8.18-8.19] y [13.17-13.19]).

Si la metodología de diseño insiste en distinguir entre entidades y vínculos, entonces (en el mejor de los casos) las dos interpretaciones serán tratadas de manera asimétrica (es decir, las consultas de "entidad" y las consultas de "vínculo" tomarán formas diferentes); en el peor de los casos, una interpretación no podrá ser soportada (es decir, será imposible de formular una clase de consulta).

Como ilustración adicional de la idea, considere la siguiente declaración de un tutorial sobre el enfoque E/R en la referencia [13.17]:

> Es común representar *inicialmente* algunos vínculos como atributos [lo que específicamente quiere decir, claves externas] durante el diseño del esquema conceptual y luego convertir estos atributos en vínculos conforme el diseño avanza y se entiende mejor.

Pero ¿qué pasa si después de un tiempo un atributo *se convierte* en una clave externa?; es decir, si la base de datos evoluciona después de estar ya en existencia por cierto tiempo. Si llevamos este argumento a su conclusión lógica, los diseños de bases de datos deben involucrar solamente vínculos, no atributos. (De hecho, esta postura tiene cierto mérito. Vea la nota a la referencia [13.18] al final del capítulo.)

Una última observación

Existen muchos otros esquemas de modelado semántico además del modelado específico E/R que hemos descrito en este capítulo. Sin embargo, la mayoría de esos esquemas tienen un parecido familiar entre sí; en particular, casi todos pueden caracterizarse como si simplemente proporcionaran una notación gráfica para representar ciertas restricciones de clave externa, además de algunos otros detalles menores. Por supuesto, dichas representaciones gráficas pueden ser útiles a manera de "panorama general", pero son demasiado simplistas para realizar el trabajo de diseño en su totalidad.* En particular (como señalamos antes) por lo regular no pueden manejar restricciones de integridad generales. Por ejemplo, ¿cómo representaría usted una dependencia de junta en un diagrama E/R?

RESUMEN

Abrimos este capítulo presentando una breve introducción a la idea general del **modelado semántico.** Cuatro grandes pasos son los que comprende, de los cuales, el primero es informal y el resto son formales:

1. Identificar conceptos semánticos útiles.
2. Crear objetos simbólicos correspondientes.
3. Crear reglas de integridad correspondientes ("metarrestricciones").
4. Crear operadores correspondientes.

---

*Es un triste comentario sobre el estado de la industria el hecho de que las soluciones simples son populares aun cuando son *demasiado* simples. Sobre estos asuntos, coincidimos con Einstein, quien alguna vez dijo: "todo debe hacerse tan simple como sea posible, *pero no más simple*".

Algunos conceptos semánticos útiles son **entidad, propiedad, vínculo** y **subtipo.** *Nota:* también subrayamos las ideas de que (a) probablemente habrá **conflictos de terminología** entre el nivel de modelado semántico (informal) y el nivel del sistema de apoyo (formal) subyacente, y (b) ¡dichos conflictos pueden causar confusión! Advertencia para el lector.

El objetivo último de la investigación en el modelado semántico es hacer un poco más inteligentes los sistemas de bases de datos. Un objetivo más inmediato es proporcionar una base para el ataque sistemático del problema del **diseño de bases de datos.** Describimos la aplicación de un modelo "semántico" en particular —el **modelo entidad/vínculo (E/R)** de Chen— para el problema del diseño.

En relación con lo anterior, vale la pena repetir la idea de que el artículo E/R original [13.5] contenía en realidad dos propuestas distintas y más o menos independientes: propuso el modelo E/R en sí, y propuso también **la técnica de elaboración de diagramas E/R.** Como señalamos en la sección 13.4, la popularidad del modelo E/R probablemente se puede atribuir más a la existencia de esa técnica de elaboración de diagramas que a cualquier otra causa. Pero la idea es que no es necesario adoptar todas las ideas del *modelo* para poder usar los *diagramas;* es muy posible utilizar los diagramas E/R como una base para *cualquier* metodología de diseño (tal vez, por ejemplo, una metodología basada en RM/T [13.6]). Los argumentos con respecto a la relativa adaptabilidad del modelado E/R, así como algún otro enfoque como base para el diseño de base de datos, a menudo parecen ignorar esta idea.

Contrastemos también las ideas del modelado semántico (y el modelo E/R en particular) con la disciplina de la normalización que describimos en los capítulos 11 y 12. La disciplina de la normalización involucra la reducción de grandes varrels en otras más pequeñas. Da por hecho que tenemos como entrada cierto número reducido de varrels grandes para producir como salida un gran número de varrels pequeñas; es decir, transforma grandes varrels en otras más pequeñas (por supuesto, aquí hablamos en términos *muy* generales). Pero la disciplina de la normalización no tiene absolutamente nada que decir sobre cómo llegamos, en primer lugar, a dichas varrels grandes. En contraste, las metodologías de arriba hacia abajo (como la que describimos en este capítulo) abordan exactamente ese problema; transforman la realidad en grandes varrels. En otras palabras, los dos enfoques —el de arriba hacia abajo y el de la normalización— *se complementan entre sí.* De esta manera, el procedimiento general de diseño va más o menos así:

1. Utilice el enfoque E/R (o algo equivalente*) para generar "grandes" varrels que represen ten entidades normales, entidades débiles, etcétera; y después.
2. Utilice las ideas de la normalización adicional para dividir dichas varrels "grandes" en otras más "pequeñas".

Sin embargo, por la índole de las explicaciones del presente capítulo, se habrá dado cuenta de que el modelado semántico en general no es en absoluto tan riguroso o claro como la disciplina de la normalización adicional que explicamos en los capítulos 11 y 12. El motivo de esta situación es que (como indicamos en la introducción a esta parte del libro) el diseño de bases de datos sigue siendo un ejercicio muy subjetivo, mas que objetivo; comparativamente, hay poco

*Nuestro enfoque preferido sería *escribir los predicados* que describen a la empresa y después transformarlos (de manera directa) en las restricciones de varrel y de base de datos, tal como lo describimos en el capítulo 8.

en cuanto a principios verdaderamente sólidos que puedan ser empleados para abordar el problema (y por supuesto, los pocos principios que existen son básicamente los que explicamos en los dos capítulos anteriores). Las ideas de este capítulo pueden ser consideradas más bien como reglas empíricas, aunque algunas parecen funcionar razonablemente bien en situaciones prácticas.

Hay una idea final que vale la pena expresar de manera explícita. Aunque todo el campo sigue siendo en cierto modo subjetivo, hay un área específica en la que las ideas del modelado semántico pueden ser muy importantes y útiles en la actualidad; es decir, el área del **diccionario de datos.** En ciertos aspectos, esta área puede ser considerada como "la base de datos del diseñador dé base de datos"; después de todo, es una base de datos en la cual se registran (entre otras cosas) decisiones de diseño de la base de datos [13.2]. El estudio del modelado semántico puede así ser útil en el diseño del sistema de diccionario, ya que identifica los tipos de objetos que el propio diccionario necesita soportar y "entender"; por ejemplo, las categorías de entidades (como las entidades normales y las débiles del modelo E/R), las reglas de integridad (como la noción del modelo E/R de la participación total contra la participación parcial en un vínculo), los supertipos y subtipos de entidades, etcétera.

EJERCICIOS

13.1 ¿Qué entiende por el término "modelado semántico"?

13.2 Identifique los cuatro grandes pasos comprendidos al definir un modelo "extendido" tal como el modelo E/R.

13.3 Defina los siguientes términos E/R:

conjunto de valores	jerarquía de tipos
entidad	propiedad
entidad débil	propiedad clave
entidad normal	supertipo, subtipo
herencia	vínculo

13.4 Proporcione ejemplos de

a. Un vínculo muchos a muchos en el que uno de los participantes sea una entidad débil;

b. Un vínculo muchos a muchos en el que uno de los participantes sea otro vínculo;

c. Un vínculo muchos a muchos que tenga un subtipo;

d. Un subtipo que tenga asociada una entidad débil que no se aplique al supertipo.

13.5 Dibuje un diagrama E/R para la base de datos de educación del ejercicio 8.10 del capítulo 8.

13.6 Dibuje un diagrama E/R para la base de datos de personal de la compañía, mostrada en el ejercicio 11.3 del capítulo 11. Use ese diagrama para derivar un conjunto adecuado de definiciones de varrels base.

13.7 Dibuje un diagrama E/R para la base de datos de pedido/entrada del ejercicio 11.4 del capítulo 11. Use ese diagrama para derivar un conjunto adecuado de definiciones de varrels base.

13.8 Dibuje un diagrama E/R para la base de datos de ventas del ejercicio 12.3 del capítulo 12. Use ese diagrama para derivar un conjunto adecuado de definiciones de varrels base.

13.9 Dibuje un diagrama E/R para la base de datos modificada de ventas, mostrada en el ejercicio 12.5 del capítulo 12. Use ese diagrama para derivar un conjunto adecuado de definiciones de varrels base.

REFERENCIAS Y BIBLIOGRAFÍA

La amplitud de la siguiente lista de referencias se debe en gran medida, a la cantidad de metodologías de diseño que compiten y que puede encontrar actualmente en el mundo de las bases de datos, tanto en la industria como en el ámbito académico. Existe muy poco consenso en este campo; el esquema E/R que explicamos en el cuerpo de este capítulo es en realidad el enfoque de mayor utilización, pero no todos están de acuerdo con él (o les agrada). De hecho, debemos señalar la idea de que los enfoques *más conocidos* no necesariamente son los *mejores* enfoques. Comentamos también que muchos de los productos disponibles comercialmente son más que sólo herramientas de diseño de base de datos; más bien, lo que hacen es generar aplicaciones completas como pantallas frontales, lógica de aplicaciones, procedimientos disparados, etcétera, así como definiciones (esquemas) de base de datos en particular.

Algunas otras referencias importantes para el material del presente capítulo son el informe ISO sobre el esquema conceptual [2.3]; el libro de Kent, *Data and Reality* [2.4]; y los libros de Ross sobre reglas de negocios [8.18-8.19].

13.1 J. R. Abrial: "Data Semantics", en J. W. Klimbie y K. L. Koffeman (eds.), *Data Base Manage ment.* Amsterdam, Países Bajos: North-Holland / Nueva York, N.Y.: Elsevier Science (1974).

Una de las primeras propuestas en el área del modelado semántico. La siguiente cita capta muy bien el sentido general del artículo (algunos dirían el tema en su totalidad): "Sugerencia para el lector: si está buscando una definición del término *semántica,* deje de leer porque no existe tal definición en este artículo."

13.2 Philip A. Bernstein: "The Repository: A Modern Vision", *Database Programming and Design* 9, No. 12 (diciembre, 1996).

Parece que al momento de la publicación de este libro hay un movimiento para sustituir el término *diccionario* por el término *repositorio.* Un sistema de repositorio es un DBMS que se especializa en la administración de metadatos, no sólo para los DBMSs sino para todo tipo de herramientas de software; para citar a Bernstein: "Herramientas para el diseño, desarrollo y distribución de software, así como herramientas para el manejo de diseños electrónicos, diseños mecánicos, sitios web y otras muchas clases de documentos formales relacionados con actividades de ingeniería." Este artículo es un tutorial sobre conceptos de repositorios.

13.3 Michael Blaha y William Premerlani: *Object-Oriented Modeling and Design for Database Ap plications.* Upper Saddle River, N.J.: Prentice-Hall (1998).

Describe a profundidad una metodología de diseño denominada Técnica de modelado de objetos (OMT). La OMT puede ser considerada como una variante del modelo E/R —sus *objetos* son básicamente *entidades* de E/R— pero cubre mucho más que el simple diseño específico de *bases de datos.* Vea también la nota a la referencia [13.32].

13.4 Grady Booch: *Object-Oriented Design with Applications.* Redwood City, Calif.: Benjamin/ Cummings(1991).

Vea la nota a la referencia [13.32].

13.5 Peter Pin-Shan Chen: "The Entity-Relationship Model—Toward a Unified View of Data", *ACM TODS I.* No. 1 (marzo, 1976). Reeditado en Michael Stonebraker (ed.), *Readings in Database Systems.* San Mateo, Calif.: Morgan Kaufmann (1988).

El artículo que presentó el modelo E/R y los diagramas E/R. Como mencionamos en el cuerpo de este capítulo, el modelo ha sido modificado y refinado a través del tiempo; en realidad, las explicaciones y definiciones dadas en este primer artículo eran bastante imprecisas, así que dichos refinamientos fueron definitivamente necesarios. (Una de las críticas al modelo E/R

siempre ha sido que los términos no parecen tener un solo significado bien definido, sino que en su lugar son interpretados de muy diversas formas. Desde luego, es cierto que todo el campo de las bases de datos está plagado de una terminología imprecisa y conflictiva, pero esta área en particular es peor que la mayoría.) Para ilustrar:

- Como dijimos en la sección 13.3, una entidad se define como "algo que puede ser identifi cado en forma distintiva" y un vínculo como "una asociación entre entidades". Entonces, la primera cuestión que surge es la siguiente: ¿un vínculo es una entidad? Es claro que un víncu lo es "algo que puede ser identificado en forma distintiva", pero las secciones posteriores del artículo parecen reservar el término "entidad" para referirse a algo que definitivamente *no* es un vínculo. Presuntamente, esta ultima es la interpretación que pretendemos; de no ser así, ¿por qué el término "entidad/vínculo"? Pero en realidad el artículo no es claro.
- Las entidades y los vínculos pueden tener *atributos* (nosotros usamos el término "propiedad" en el cuerpo del capítulo). Una vez más, el artículo es ambivalente con respecto al significado del término; al principio define un atributo como una propiedad que no es la clave primaria ni componente alguno de ella (compare la definición relacional), pero más adelante usa el tér mino en el sentido relacional.
- Damos por hecho que la clave primaria de un vínculo es la combinación de las claves exter nas que identifican a las entidades involucradas en el vínculo (sin embargo, no empleamos el término "clave externa"). Esta suposición sólo es apropiada para vínculos muchos a muchos, y no siempre. Por ejemplo, considere la varrel VPF {V#,P#,FECHA,CANT), la cual repre senta los envíos de ciertas partes por ciertos proveedores en ciertas fechas; suponga que el mismo proveedore puede enviar la misma pieza más de una vez, pero no más de una vez en la misma fecha. Entonces, la clave primaria (o por lo menos la única clave candidata) es la combinación {V#,P#,FECHA}; aunque podríamos optar por considerar a los proveedores y a las partes (pero no a las fechas) como entidades.

13.6 E. F. Codd: "Extending the Database Relational Model to Capture More Meaning", *ACM TODS 4,* No. 4 (diciembre, 1979).

En este artículo, Codd presentó una versión "extendida" del modelo relacional a la que llamó RM/T. El RM/T aborda algunos de los mismos aspectos que el modelo E/R pero está definido de manera más cuidadosa. Algunas de las diferencias inmediatas entre ambos son las siguientes. Primero, el RM/T no hace distinciones innecesarias entre entidades y vínculos (un vínculo es considerado simplemente como una clase especial de entidad). Segundo, los aspectos estruc- turales y de integridad del RM/T son más amplios y están definidos de manera más precisa que los del modelo E/R. Tercero, el RM/T incluye sus propios operadores especiales, además de los operadores del modelo relacional básico (aunque queda mucho por hacer en esta área).

A grandes rasgos, el RM/T funciona de la siguiente forma:

- Primero, las entidades (incluyendo los "vínculos") son representados mediante *relaciones E* y *P,** que son formas especiales de la relación n-aria general. Las relaciones E se usan para registrar el hecho de que existen ciertas entidades, mientras que las relaciones P se usan para registrar ciertas propiedades de dichas entidades.
- Segundo, puede existir una variedad de vínculos entre las entidades; por ejemplo, los tipos de entidad *A y B* podrían enlazarse juntos en una **asociación** (el término del RM/T para una vínculo muchos a muchos) o el tipo de entidad *Y* podría ser un **subtipo** del tipo de entidad *X.*

---

*O, más bien, *varrels* E y P

El RM/T incluye una estructura de **catálogo** formal mediante la cual dichos vínculos pueden hacerse del conocimiento del sistema; el sistema es entonces capaz de hacer cumplir las diversas **restricciones de integridad** que están implicadas por la existencia de tales vínculos.

- Tercero, se proporcionan diversos **operadores** de alto nivel para facilitar la manipulación de los distintos objetos RM/T (relaciones E, relaciones P, relaciones de catálogo, etcétera).

Al igual que en el modelo E/R, el RM/T incluye equivalentes de todas las construcciones (entidad, propiedad, vínculo, subtipo) que listamos en la figura 13.1. En particular, proporciona un **esquema de** clasificación **de entidades** (el cual constituye en muchos sentidos el aspecto más significativo —o por lo menos, el inmediatamente más visible— de todo el modelo), de acuerdo con el cual las entidades están divididas en tres categorías; es decir, *núcleos, características y asociaciones:*

- **Núcleos.** Las entidades núcleo son aquellas que tienen *existencia independiente;* son "de lo que en realidad trata la base de datos". En otras palabras, los núcleos son entidades que ni son características ni asociativas (vea abajo).
- **Características.** Una entidad característica es aquella cuyo propósito principal es describir o "caracterizar" a alguna otra entidad. Las características son *dependientes de la existencia* de la entidad que describen. La entidad descrita puede ser de núcleo, característica o asociativa.
- **Asociaciones.** Una entidad asociativa es la que representa un *vínculo muchos a muchos* (o muchos a muchos a muchos, etcétera) entre dos o más entidades. Las entidades asociadas pueden ser cada una de núcleo, característica o asociativa.

Además:

- Las entidades (independientemente de su clasificación) pueden también tener **propiedades.**
- En particular, cualquier entidad (una vez más, independientemente de su clasificación) puede tener una propiedad que **designe** alguna otra entidad relacionada. Una designación representa un vínculo muchos a uno entre dos entidades. *Nota:* Las designaciones no fueron explicadas en el artículo original [13.6] pero fueron agregadas posteriormente.
- Los **supertipos** y **subtipos** de entidad son soportados. Si *Y* es un subtipo de *X,* entonces *Y* es un núcleo, una característica o una asociación, dependiendo de si *X* es un núcleo, una característica o una asociación.

Podemos relacionar (en cierta forma general) los conceptos anteriores con sus equivalentes del modelo E/R, como sigue: un núcleo corresponde a una "entidad normal" E/R, una característica a una "entidad débil" E/R, y una asociación a un "vínculo" E/R (sólo de la variedad muchos a muchos).

Nota: Otro término que encontramos en ocasiones en la literatura, *dominio primario,* fue definido también por vez primera en este artículo. Un **dominio primario** es un dominio en el cual está definida al menos una clave primaria de un solo atributo (es decir, no compuesta). Por ejemplo, en el caso de proveedores y partes, los dominios primarios son V# y P#.

Además de los aspectos que describimos brevemente arriba, el RM/T incluye también soporte para (a) *sustitutos* (vea la referencia [13.16]), (b) la dimensión *tiempo* (vea el capítulo 22), y (c) varias clases de *agregación de datos* (vea las referencias [13.35 y 13.36]).

13.7 C. J. Date: "A Note on One-to-One Relationships", en *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Una amplia exposición del problema de los vínculos uno a uno, el cual resulta ser más complicado de lo que podría parecer a primera vista.

13.8 C. J. Date: "Entity/Relationship Modeling and the Relational Model", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

13.9 C. J. Date: "Don't Encode Information into Primary Keys!", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Presenta una serie de argumentos informales contra lo que a veces se denomina "claves inteligentes". Vea también la referencia [13.10] para algunas recomendaciones relacionadas con respecto a las claves externas.

13.10 C. J. Date: "Composite Keys", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Para parafrasear: "resume los argumentos a favor y en contra de la inclusión de [claves] compuestas en el diseño de una base de datos relacional y... ofrece algunas recomendaciones". En particular, el artículo muestra que las claves sustitutas [13.16] *no siempre* son una buena idea.

13.11 C. J. Date: "A Database Design Dilemma?", en el sitio web *DBP&D www.dbpd.com* (enero, 1999). Vea también el apéndice B de la referencia [3.3].

A primera vista, un tipo de entidad dado —digamos, empleados— podría ser representado en un sistema relacional ya sea mediante un *tipo* empleados (es decir, un dominio) o mediante una *varrel* empleados. Este breve artículo ofrece una guía sobre cómo elegir entre las dos opciones.

13.12 C. J. Date: "Subtables and Supertables" (en dos partes), en el sitio web *DBP&D www.dbpd.com* (por aparecer a finales del 2000 o principios del 2001). Vea también el apéndice D de la referencia [3.3].

A menudo creemos que la herencia de tipos de entidad debe ser tratados dentro de un contexto relacional por medio de lo que llamamos "subtablas y supertablas" (el subtipo de entidad corresponde a una "subtabla" y el supertipo de entidad a una "supertabla"). Por ejemplo, al momento de la publicación de este libro, SQL3 soportaba dicho enfoque (vea el apéndice B), así como también ciertos productos. Este artículo argumenta fuertemente contra dicha idea.

13.13 Ramez Elmasri y Shamkant B. Navathe: *Fundamentals of Database Systems* (2a. edición). Redwood City, Calif.: Benjamin/Cummings (1994).

Este libro de texto general sobre administración de bases de datos incluye dos capítulos completos (de un total de 25) sobre el uso de técnicas E/R para el diseño de bases de datos.

13.14 David W. Embley: *Object Database Development: Concepts and Principles.* Reading, Mass.: Addison-Wesley (1998).

Presenta una metodología de diseño basada en el OSM (Modelo de Sistemas Orientados a Objetos). Partes del OSM se asemejan al ORM [13.17-13.19].

13.15 Candace C. Fleming y Barbara von Hallé: *Handbook of Relational Database Design.* Reading, Mass.: Addison-Wesley (1989).

Una buena guía pragmática para el diseño de bases de datos en un sistema relacional, con ejemplos específicos basados en el producto DB2 de IBM y en el producto DBC/1012 de Teradata (ahora NCR). Aborda los aspectos de diseño lógico y físico; aunque el libro usa el término "diseño lógico" para referirse a lo que nosotros llamaríamos "diseño relacional", así como el término "diseño relacional" para incluir por lo menos algunos aspectos de lo que nosotros llamaríamos "diseño físico".

13.16 P. Hall, J. Owlett y S. J. P. Todd: "Relations and Entities", en G. M. Nijssen (ed.), *Modelling in Data Base Management Systems.* Amsterdam, Países Bajos: North-Holland / Nueva York, N.Y.: Elsevier Science (1975).

El primer artículo en tratar con detalle el concepto de **claves sustituías** (el concepto fue incorporado más adelante en el RM/T [13.6]). Las claves sustitutas son claves en el sentido relacional usual pero tienen las siguientes propiedades específicas:

- Siempre comprenden exactamente un atributo.
- Sus valores sirven *solamente* como sustitutos (de ahí su nombre) para las entidades que repre sentan. En otras palabras, dichos valores sirven simplemente para representar el hecho de que existen las entidades correspondientes; no llevan información alguna o significado adicionales.
- Al insertar una nueva entidad dentro de la base de datos, le es asignado un valor de clave **susti tuía** que nunca ha sido usado anteriormente y nunca será usado de nuevo, incluso si la enti dad en cuestión es eliminada después.

De manera ideal, los valores de claves sustitutas serían generados por el sistema, pero el hecho de ser generados por el sistema o por el usuario no tiene nada que ver con la idea básica de las claves sustitutas como tales.

Vale la pena enfatizar que los sustitutos *no son* (como parecen pensar algunos autores) lo mismo que los "IDs de tupia", ya que —para decir lo obvio— los IDs de tupia identifican tupias y los sustitutos identifican entidades, y en realidad no hay nada como una correspondencia de uno a uno entre las tupias y las entidades (piense en particular, en los IDs de las tupias derivadas). Además, los IDs de tupia tienen connotaciones de rendimiento, mientras que los sustitutos no; por lo regular suponemos que el acceso a una tupia por vía de un ID de tupia es rápido (nosotros aquí suponemos que las tupias —por lo menos las tupias de relaciones base— se transforman de manera bastante directa al almacenamiento físico, como es de hecho el caso en la mayoría de los productos actuales). Además, los IDs de tupia se concilian por lo regular desde el usuario, mientras que los sustitutos no deben hacerlo (debido al *principio de información);* en otras palabras, no es posible almacenar un ID de tupia como un valor de atributo, en tanto que en realidad es posible almacenar un sustituto como un valor de atributo.

En resumen: Los sustitutos son un concepto lógico; los IDs de tupia son un concepto físico.

13.17 Terry Halpin: *Conceptual Schema and Relational Database Design* (2a edición). Sydney, Australia: Prentice-Hall of Australia Pty., Ltd. (1995).

Un tratamiento detallado del ORM (vea más adelante las notas a las dos siguientes referencias).

13.18 Terry Halpin: "Business Rules and Object-Role Modeling", *DBP&D* 9, No. 10 (octubre, 1996).

Una excelente introducción al **modelado objeto-papel** (ORM) [13.17]. Halpin comienza por observar que "[a diferencia del] modelado E/R —el cual tiene docenas de dialectos diferentes—, el ORM tiene sólo unos cuantos dialectos con diferencias menores". *{Nota:* Uno de esos dialectos es NIAM [13.29]). Al ORM también se le conoce como modelado *basado en hechos,* debido a que lo que el diseñador hace es escribir —ya sea en lenguaje natural o en una notación gráfica especial— una serie de hechos *elementales* (o más bien, *tipos* de hechos) que en su conjunto caracterizan la empresa a modelar. Ejemplos de estos tipos de hechos podrían ser:

- Cada empleado tiene como máximo un Nomemp.
- Cada empleado reporta como máximo a un Empleado.
- Si el empleado *el* reporta al empleado *e2,* entonces no puede ser que el empleado *e2* reporte al empleado *el.*
- Ningún empleado puede dirigir y evaluar el mismo proyecto.

Como puede ver, los tipos de hechos son en realidad *predicados* o reglas de negocios; como sugiere el título del artículo de Halpin, el ORM está contagiado del espíritu del enfoque de diseño de base de datos preferido por quienes defienden las "reglas de negocios" [8.18-8.19], y de hecho también por mí. En general, los hechos especifican *papeles* que desempeñan los *objetos*

en los vínculos (de ahí el nombre de "modelado objeto-papel"). Observe que (a) aquí los "objetos" significan en realidad entidades, no objetos en el sentido especial que describimos en la parte VI de este libro y (b) los vínculos no son necesariamente binarios. Sin embargo, los hechos son *elementales;* no pueden ser descompuestos en otros más pequeños. *Nota:* La idea de que la base de datos debe contener sólo hechos elementales (o *irreducibles)* al nivel conceptual fue propuesta antes por Hall, Owlett y Todd [13.16].

Observe que el ORM no tiene el concepto de "atributos". Como consecuencia, los diseños ORM son conceptualmente más simples y sólidos que sus contrapartes en el E/R, como muestra el artículo (al respecto, vea también la nota a la referencia [13.19]). Sin embargo, los atributos pueden aparecer y aparecen en diseños E/R o SQL que son generados (automáticamente) a partir de un diseño ORM.

El ORM enfatiza también el uso de "hechos de ejemplo" (es decir, *muestras* de hechos de ejemplo, que nosotros llamaríamos *proposiciones)* como una forma para permitir al usuario final la validación del diseño. La afirmación es que dicho enfoque es directo con el modelado basado en hechos y mucho menos directo con el modelado E/R.

Por supuesto, hay muchas formas lógicamente equivalentes de describir una empresa dada, y por ello hay muchos esquemas ORM lógicamente equivalentes. Por lo tanto, el ORM incluye un conjunto de *reglas de transformación* que permiten transformar entre sí esquemas lógicamente equivalentes, de modo que una herramienta ORM puede realizar cierta optimización sobre el diseño tal como lo especificó el diseñador humano. También puede (como mencionamos anteriormente) generar un esquema E/R o un esquema SQL a partir de un esquema ORM, y puede (por el contrario) generar un esquema ORM a partir de un esquema E/R o SQL. Dependiendo del DBMS de destino, un esquema SQL generado puede incluir restricciones declarativas (al estilo de SQL/92), o puede implementar dichas restricciones a través de procedimientos almacenados o disparados. Por cierto, observe que con respecto a las restricciones —a diferencia del modelo E/R— el ORM incluye *por definición* "un lenguaje rico para expresar restricciones" . (Sin embargo, Halpin admite en la referencia [13.19] que no todas las reglas de negocios pueden ser expersadas en la notación *gráfica* ORM; sigue siendo necesario el texto para este fin.)

Por último, un esquema ORM puede (desde luego) ser considerado como una vista abstracta —de alto nivel— de la base de datos (de hecho, argumentaríamos que está cerca de ser una vista *relacional* pura, quizás en cierto modo disciplinada). Como tal, puede ser **consultada directamente.** Vea la nota a la referencia [13.19] que sigue.

13.19 Terry Halpin: "Conceptual Queries", *Data Base Newsletter 26,* No. 3 (marzo-abril, 1998).

Para citar el resumen: "Formular consultas no triviales en lenguajes relaciónales como SQL o QBE puede resultar desalentador para los usuarios finales. *ConQuer,* un nuevo lenguaje de consulta conceptual basado en el modelado objeto-papel (ORM), permite a los usuarios proponer consultas en una forma realmente fácil de entender... Este artículo resalta las ventajas de [dicho lenguaje] sobre los lenguajes tradicionales de consulta, para especificar consultas y reglas de negocios".

Entre otras cosas, el artículo explica una consulta de ConQuer junto con las siguientes líneas:

■*S* Empleado
H— maneja Automóvil
H— trabaja para División *S*

("Obtener los empleados y la división para aquellos empleados que manejan automóvil".) Si los empleados pueden manejar cualquier cantidad de automóviles pero trabajar sólo para una división, el diseño SQL subyacente involucraría dos tablas y el código SQL generado luciría similar al siguiente:

```
SELECT DISTINCT X1.EMP#, X1.DIVISION*
FROM  EMPLEADO AS X1, MANEJA AS X2
WHERE  X1.EMP# = X2.EMP# ;
```

Ahora, suponga que es posible que los empleados trabajen al mismo tiempo para varias divisiones. Entonces, el diseño SQL subyacente cambiaría para involucrar tres tablas en lugar de dos y el código SQL generado cambiaría también:

```
SELECT DISTINCT X1.EMP#, X3.DIVISION*
FROM  EMPLEADO AS X1, MANEJA AS X2, TRABAJA_PARA AS X3
WHERE  X1.EMP* = X2.EMP# AND X1.EMP* • X3.EMP# ;
```

Sin embargo, la formulación de ConQuer permanece sin cambios.

Como ilustra el ejemplo anterior, un lenguaje como ConQuer puede ser considerado como una forma particularmente sólida de independencia lógica de los datos. Sin embargo, para poder explicar esta observación, primero necesitamos definir de algún modo la arquitectura ANSI/SPARC [2.1-2.2]. En el capítulo 2 dijimos que la independencia lógica de los datos significaba independencia de cambios en el esquema conceptual, ¡pero toda la idea del ejemplo anterior es que no ocurren cambios en el esquema conceptual! El problema es que los productos SQL actuales no soportan adecuadamente un esquema conceptual, en su lugar soportan un esquema *SQL*. Y podemos considerar que ese esquema SQL está ubicado en un nivel intermedio entre el verdadero nivel conceptual y el nivel interno o físico. Si una herramienta ORM nos permite definir un verdadero esquema conceptual y después asociarlo a un esquema SQL, entonces ConQuer puede proporcionar independencia de cambios para ese esquema SQL (al hacer, por supuesto, los cambios correspondientes a la transformación).

En el artículo no está claro qué límites podría haber en el poder expresivo de ConQuer. Halpin no aborda directamente esta cuestión; sin embargo, sí dice (en forma un poco preocupante) que "en forma ideal, el lenguaje debe permitir a su aplicación formular cualquier pregunta relevante; en la práctica, algo menos que este ideal es aceptable". También declara que "la característica más poderosa [de ConQuer]... es su capacidad para realizar correlaciones de complejidad arbitraria", y ofrece el siguiente ejemplo:

■^Empleado 1

H—vive en la Ciudad 1
H— nació en Países
H— supervisa al Empleado2
H— vive en la Ciudad 1
+- nació en País2 O País 1

("Obtener los empleados que supervisan a un empleado que vive en la misma ciudad que el supervisor pero que nació en un país diferente al del supervisor".) Como dice su autor: "¡Intente hacer esto en SQL!"

Por último, con respecto a ConQuer y las reglas de negocios, Halpin tiene esto que decir: "Aunque la notación gráfica del ORM puede captar más reglas de negocios [que los enfoques E/R], aún es necesario complementarlo por medio de un lenguaje textual [para expresar ciertas restricciones]. La investigación está en camino de adaptar a ConQuer para este fin."

13.20 M. M. Hammer y D. J. McLeod: "The Semantic Data Model: A Modelling Mechanism for Database Applications", Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Texas (mayo-junio, 1978).

El Modelo de Datos Semántico (SDM) representa otra propuesta de un formalismo de diseño de bases de datos. Al igual que el modelo E/R, se concentra en los aspectos estructural y (hasta cierto grado) de integridad, y tiene poco o nada qué decir con respecto a los aspectos de manipulación. Vea también las referencias [13.21] y [13.24].

13.21 Michael Hammer y Dennis McLeod: "Database Description with SDM: A Semantic Database Model", *ACM TODS* 6, No. 3 (septiembre, 1981).

Vea la nota a la referencia [13.20].

13.22 Richard Hull y Roger King: "Semantic Database Modeling: Survey, Applications, and Re search Issues", *ACM Comp. Surv. 19,* No. 3 (septiembre, 1987).

Un amplio tutorial sobre el campo del modelado semántico y aspectos relacionados de finales de los años ochenta. Este artículo es un buen punto para iniciar una investigación más profunda de los aspectos y problemas de investigación que rodean las actividades del modelado semántico. Vea también la referencia [13.31].

13.23 Ivar Jacobson *et al: Object-Oriented Software Engineering* (edición revisada). Reading, Mass.: Addison-Wesley (1994).

Describe una metodología de diseño denominada OOSE (Ingeniería de Software Orientado a Objetos). Al igual que la OMT [13.3], al menos las partes de base de datos de OOSE pueden ser consideradas como una variante del modelo E/R (al igual que con la OMT, los *objetos* OOSE son básicamente entidades E/R). Vale la pena observar la siguiente cita: "La mayoría de los instrumentos usados en la industria actual para desarrollar sistemas tanto de información como técnicos, están basados en una descomposición funcional o conducida por datos del sistema. Estos enfoques difieren en muchas formas del enfoque que toman los métodos orientados a objetos en donde los datos y las funciones están altamente integrados." Nos parece que aquí Jacobson hace énfasis en una incongruencia importante entre el pensamiento de objetos y el de base de datos. Las bases de datos —por lo menos las compartidas de propósito general, que son por mucho el principal enfoque de la comunidad de bases de datos— *supuestamente* están divorciadas de las "funciones"; *supuestamente* están diseñadas en forma separada de las aplicaciones que las usan. De ahí que nos parezca que el término "base de datos", tal como es empleado en la comunidad de objetos, en realidad significa una base de datos que es *específica de la aplicación,* no una compartida y de propósito general.

Vea también (a) la nota a la referencia [13.32], y (b) la explicación de bases de datos de objetos en el capítulo 24.

13.24 D. Jagannathan *et al.:* "SIM: A Database System Based on the Semantic Data Model", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. (junio, 1988).

Describe un producto DBMS comercial basado en "un modelo de datos semántico similar al" Modelo Semántico de Datos propuesto por Hammer y McLeod en la referencia [13.20].

13.25 Warren Keuffel: "Battle of the Modeling Techniques: A Look at the Three Most Popular Modeling Notations for Distilling the Essence of Data", *DBMS 9,* No. 9 (agosto, 1996).

Decimos que las "tres notaciones más importantes" son el modelado E/R, el NIAM (Método de Análisis de Información en Lenguaje Natural) de Nijssen [13.29] y el SOM (Modelado de Objetos Semánticos). Keuffel afirma que el modelado E/R es el "abuelo" de los otros dos, pero critica la falta de una base formal; como él dice, todas las entidades, vínculos y atributos (es decir, las propiedades) son "descritas sin hacer referencia a cómo fueron descubiertas". El NIAM es mucho más riguroso; cuando sus reglas son seguidas fielmente, los diseños conceptuales resultantes "poseen mucha más integridad" que los diseños producidos empleando otras metodologías, aunque "algunos desarrolladores encuentren el rigor de NIAM demasiado limitante" (!). En cuanto al SOM, "se parece al modelado E/R... con definiciones de entidades, atributos y vínculos [en forma similar] articuladas vagamente"; sin embargo, difiere del modelado E/R en que soporta *atributos de grupo* (es decir, grupos repetitivos), lo que permite que un "objeto" (es decir una entidad) contenga a otros. (El modelado E/R permite que las entidades contengan grupos de *atributos* repetitivos, pero no otras *entidades.)*

13.26 Heikki Mannila y Kari-Jouko Raihá: *The Design of Relational Databases.* Reading, Mass.: Addison-Wesley (1992).

Para citar el prefacio, este libro es "un libro de texto de nivel poslicenciatura y una referencia sobre el diseño de bases de datos relaciónales". Por una parte, cubre tanto la teoría de la dependencia como la normalización, y por otra, el enfoque E/R; en cada caso desde una perspectiva bastante formal. La siguiente lista (incompleta) de títulos de los capítulos ofrece una idea del alcance del libro:

- Principios de diseño;
- Restricciones de integridad y dependencias;
- Propiedades de los esquemas relaciónales;
- Axiomatizaciones para dependencias;
- Algoritmos para problemas de diseño;
- Transformaciones entre diagramas E/R y esquemas de bases de datos relaciónales;
- Transformaciones de esquemas;
- Uso de bases de datos de ejemplo en el diseño.

Las técnicas descritas en el libro han sido implementadas por los autores en forma de una herramienta disponible comercialmente llamada Design By Example.

13.27 Terry Moriarty: *Enterprise View* (columna regular), *DBP&D 10,* No. 8 (agosto, 1997).

Describe una herramienta de diseño y desarrollo de aplicaciones comerciales denominada Usoft *(www.usoft.com)* que permite definir reglas de negocios utilizando una sintaxis al estilo de SQL, y emplea dichas reglas para generar la aplicación (incluyendo la definición de la base de datos).

13.28 G. M. Nijssen, D. J. Duke y S. M. Twine: "The Entity-Relationship Data Model Considered Harmful", Proc. 6th Symposium on Empirical Foundations of Information and Software Sciences, At lanta, Ga. (octubre, 1988).

"¿Se considera perjudicial al modelo E/R?" Pues bien, parece que tiene mucho por responder, incluyendo:

- La confusión sobre los tipos y las varrels (vea la explicación de **el primer gran error garra fal** en el capítulo 25);
- El extraño asunto de las "subtablas y supertablas" [13.12];
- Una muy extendida falla para apreciar *el principio de relatividad de las bases de datos* (vea el capítulo 9);
- La confusión sobre las entidades y vínculos mismos, como explicamos en el presente capítulo.

A la letanía anterior se agrega la referencia [13.28]. Para ser más específicos, afirma que el modelo E/R:

- Ofrece muchas formas que se traslapan para representar la estructura de datos y complica así indebidamente el proceso de diseño;
- No proporciona una guía sobre cómo seleccionar entre representaciones alternativas y de hecho puede requerir que los diseños existentes sean modificados innecesariamente si las cir cunstancias cambian;
- Ofrece muy pocas formas de representar la integridad de los datos y por lo tanto, hace im posibles ciertos aspectos del proceso de diseño ("[es cierto que] las restricciones pueden ser expresadas formalmente en una notación más general... [como, por ejemplo] la lógica de pre dicados; [pero] decir que ésta es una excusa razonable para omitir [las restricciones] del pro-

pió modelo de datos, es como decir que un lenguaje de programación es adecuado [incluso aunque] le obligue a llamar rutinas de lenguaje ensamblador para implementar todas esas cosas que no puede usted expresar en el lenguaje mismo");

- Al contrario de la opinión popular, no sirve como un buen vehículo para la comunicación entre los usuarios finales y los profesionales de bases de datos; y
- Viola *el **principio de conceptuación:*** "Un esquema conceptual debe... incluir [sólo] aspectos conceptualmente relevantes, tanto estáticos como dinámicos, del universo de discurso, para excluir todos los aspectos (internos y externos) de representación de datos, organización física de datos y acceso, [así como] todos los aspectos de representación particular externa del usua rio, como formatos de mensajes, estructuras de datos, etcétera" [2.3]. De hecho, los autores sugieren que el modelo E/R es "esencialmente sólo una reencarnación" del antiguo modelo de red CODASYL (vea el capítulo 1). "¿Pudo ser esta fuerte desviación hacia las estructuras de implementación la razón principal por la que el modelo E/R ha recibido tan amplia aceptación en la comunidad [de bases de datos] profesional?"

El artículo identifica también diversas debilidades adicionales del modelo E/R en el nivel de detalle. Propone entonces la metodología alternativa NIAM [13.29] como la forma a seguir. En particular, subraya la idea de que NIAM no incluye la distinción innecesaria del E/R entre atributos y vínculos.

13.29 T. W. Olle, H. G. Sol y A. A. Verrijn-Stuart (eds.): *Information Systems Design Methodolo gies: A Comparative Review.* Amsterdam, Países Bajos: North-Holland / Nueva York, N. Y.: Elsevier Science (1982).

Las memorias de una conferencia del Grupo de trabajo IFIP 8.1. Describen unas 13 metodologías diferentes que son aplicadas a un problema de prueba de desempeño estándar. Una de las metodologías incluidas es NIAM (vea la referencia [13.28]); el artículo en cuestión debe ser uno de los primeros sobre el enfoque NIAM. El libro incluye también exámenes de algunos de los enfoques propuestos, incluyendo en particular una vez más al NIAM.

13.30 M. P. Papazoglou: "Unraveling the Semantics of Conceptual Scnemas", *CACM 38,* No. 9 (septiembre, 1995).

Este artículo propone un enfoque a lo que podríamos llamar *consultas de metadatos;* es decir, consultas con respecto al significado (a diferencia de los valores) de los datos en la base de datos o, en otras palabras, consultas con respecto al esquema conceptual mismo. Un ejemplo de una consulta así podría ser "¿Qué es un empleado permanente?"

13.31 Joan Peckham y Fred Maryanski: "Semantic Data Models", *ACM Comp. Surv. 20,* No. 3 (septiembre, 1988).

Otra inspección tutorial (vea también la referencia [13.22]).

13.32 Paul Reed: "The Unified Modeling Language Takes Shape", *DBMS 11,* No. 8 (julio, 1998).

El UML (Lenguaje de Modelado Unificado) es otra notación gráfica para soportar la tarea del diseño y desarrollo de aplicaciones (en otras palabras, le permite desarrollar aplicaciones dibujando imágenes). También puede ser usado para desarrollar esquemas de SQL. *Nota:* Es probable que el UML se vuelva importante comercialmente, en parte porque fue adoptado por el OMG (Grupo de Administración de Objetos) y tiene en general un fuerte sentido hacia los objetos. Ya hay varios productos comerciales que lo soportan.

El UML soporta el modelado tanto de datos como de procesos (en este aspecto, va más allá que el modelado E/R), pero no parece tener mucho qué decir con respecto a las restricciones de integridad. (La sección de la referencia [13.32] titulada "De los modelos al código: Reglas de

negocios" no menciona en absoluto el término *declarativo*. Más bien, se concentra en la generación de *código de procedimientos de las aplicaciones* para implementar "procesos". Aquí tenemos una cita directa: "El UML formaliza lo que quienes practican con objetos conocen desde hace mucho: los objetos de la realidad se modelan mejor como entidades contenidas en sí mismas que contienen tanto datos como funcionalidad". Y en otra parte: "Desde una perspectiva histórica, es evidente que la separación formal de datos y funciones se ha traducido, en el mejor de los casos, en gran parte de nuestros frágiles esfuerzos de desarrollo de software." Estas observaciones podrían parecer válidas desde la perspectiva de una aplicación, pero no está del todo claro si lo son desde una perspectiva de base de datos. Por ejemplo, vea la referencia [24.29].) El UML surgió a partir de un trabajo previo de Booch sobre el "Método Booch" [13.4], de Rumbaugh sobre OMT [13.3] y de Jacobson sobre OOSE [13.23]. Booch, Rumbaugh y Jacobson han producido recientemente una serie de libros sobre UML, que sin duda se convertirán en referencias definitivas: *The Unified Modeling Language User Guide, The Unified Modeling Language Reference Manual* y *The Unified Software Development Process;* todos ellos publicados por Addison-Wesley en 1999.

13.33 H. A. Schmid y J. R. Swenson: "On the Semantics of the Relational Data Base Model", Proc. 1975 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif, (mayo, 1975).

Este artículo propuso un "modelo semántico básico" que antecedió al trabajo de Chen sobre el modelo E/R [13.5], pero que fue de hecho muy similar a ese modelo (excepto —por supuesto— en la terminología; Schmid y Swenson utilizan *objeto independiente, objeto dependiente* y *asociación* en lugar de los términos de Chen *entidad normal, entidad débil* y *vínculo,* respectivamente).

13.34 J. F. Sowa: *Conceptual Structures: Information Processing in Mind and Machine.* Reading, Mass.: Addison-Wesley (1984).

Este libro no trata de sistemas de bases de datos en forma específica, sino más bien sobre el problema general de la representación y procesamiento del conocimiento. Sin embargo, algunas partes son importantes directamente para el tema del presente capítulo. (Las observaciones que siguen están basadas en una presentación en vivo de Sowa alrededor de 1990 sobre la aplicación de "estructuras conceptuales" para el modelado semántico.) Un problema principal con los diagramas E/R y formalismos similares es que son estrictamente menos poderosos que la lógica formal. Como consecuencia, son incapaces de tratar con ciertas características importantes del diseño —en particular, todo lo que comprende cuantificadores, lo cual incluye la mayoría de las restricciones de integridad— que *pueden* ser manejadas por la lógica formal. (Los cuantificadores los inventó Frege en 1879, lo cual hace a los diagramas E/R "una clase de lógica anterior a 1879".) Pero la lógica formal tiende a ser difícil de leer; como dice Sowa: "el cálculo de predicados es el lenguaje ensamblador de la representación del conocimiento". Los *grafos conceptuales* son una notación gráfica legible y rigurosa que puede representar a toda la lógica. Por lo tanto están (afirma Sowa) mucho mejor adaptados a la actividad del modelado semántico que los diagramas E/R y similares.

13.35 J. M. Smith y D. C. P. Smith: "Database Abstractions: Aggregation", *CACM 20,* No. 6 (junio, 1977).

Vea la referencia [13.36] que sigue.

13.36 J. M. Smith y D. C. P. Smith: "Database Abstractions: Aggregation and Generalization", ACM *TODS 2,* No. 2 (junio, 1977).

Las propuestas de estos dos artículos [13.35 y 13.36] tuvieron una influencia significativa en el RM/T [13.6], en especial en el área de subtipos y supertipos de entidades.

13.37 Veda C. Storey: "Understanding Semantic Relationships", *The VLDB Journal* 2, No. 4 (oc tubre, 1993).

Para citar el resumen: "Los modelos de datos semánticos han sido desarrollados [en la comunidad de bases de datos] por medio de abstracciones como [los subtipos], la agregación y la asociación. Además de estos vínculos bien conocidos, han sido identificados diversos vínculos semánticos adicionales por investigadores de otras disciplinas como la lingüística, la lógica y la sicología cognitiva. Este artículo explora algunos de [estos últimos] vínculos y explica... su impacto en el diseño de bases de datos".

13.38 B. Sundgren: "The Infological Approach to Data Bases", en J. W. Klimbie y K. L. Koffeman (eds.), *Data Base Management.* Amsterdam, Países Bajos: North-Holland / Nueva York, N.Y.: Elsevier Science (1974).

El "enfoque infológico" fue uno de los primeros esquemas de modelado semántico en ser desarrollado. En Escandinavia ha sido utilizado exitosamente por muchos años para el diseño de bases de datos.

13.39 Dan Tasker: *Fourth Generation Data: A Guide to Data Analysis for New and Old Systems.* Sydney, Australia: Prentice-Hall of Australia Pty., Ltd. (1989).

Una buena guía pragmática para el diseño de bases de datos, que hace énfasis en los elementos de datos individuales (es decir, los *dominios).* Los elementos de datos están divididos en tres clases básicas: etiqueta, cantidad y descripción. Los elementos *etiqueta* se refieren a las entidades; en términos relaciónales, corresponden a las claves primaria y externa. Los elementos *cantidad* representan cantidades, medidas o posiciones sobre una escala (posiblemente una escala de tiempo) y están sujetos a la manipulación aritmética usual. Los elementos *descripción* son todo los demás. (Por supuesto, hay mucho más sobre el esquema de clasificación de lo que puede sugerir este esbozo.) El libro continúa manejando cada clase con un detalle considerable. Las explicaciones no siempre son "relacionalmente puras" —por ejemplo, el uso de Tasker del término "dominio" no coincide por completo con el uso relacional del mismo— aunque el libro sí contiene una gran cantidad de consejos prácticos sólidos.

13.40 Toby J. Teorey y James P. Fry: *Design of Database Structures.* Englewood Cliffs, N.J.: Pren tice-Hall (1982).

Un libro de texto sobre todos los aspectos del diseño de bases de datos. El libro está dividido en cinco partes: Introducción, Diseño conceptual, Diseño de implementación (es decir, la transformación del diseño conceptual en construcciones que un DBMS específico pueda entender), Diseño físico y Aspectos especiales del diseño.

13.41 Toby J. Teorey, Dongqing Yang y James P. Fry: "A Logical Design Methodology for Re lational Databases Using the Extended Entity-Relationship Model", *ACM Comp. Surv. 18,* No. 2 (junio, 1986).

El "modelo E/R extendido" del título de este artículo agrega el soporte de jerarquías de tipos de entidad, valores nulos (vea el capítulo 18) y vínculos que involucran a más de dos participantes.

13.42 Toby J. Teorey: *Database Modeling and Design: The Entity-Relationship Approach* (3a. edición). San Francisco, Calif.: Morgan Kaufmann (1998).

Un libro de texto más reciente sobre la aplicación del modelo E/R y conceptos E/R "extendidos" [13.41] al diseño de bases de datos.

PARTE IV

ADMINISTRACIÓN DE TRANSACCIONES

Esta parte del libro consta de dos capítulos. Los temas de estos capítulos (recuperación y concurrencia) están muy interrelacionados, ya que ambos aspectos forman parte del tema más amplio de la *administración de transacciones.* Sin embargo, por razones pedagógicas es preferible tratarlos en forma separada en la medida de lo posible.

La recuperación y la concurrencia (o mejor dicho, los *controles* para la recuperación y la concurrencia) están relacionadas con la cuestión general de la **protección de los datos;** es decir, la protección contra la pérdida o daño de la información que está en la base de datos. En particular tienen que ver con problemas como los siguientes:

- El sistema puede abortar cuando está ejecutando algún programa, con lo que dejaría a la base de datos en un estado desconocido.
- Dos programas que están ejecutándose al mismo tiempo ("en forma concurrente") pueden interferir entre sí y en consecuencia, producir resultados incorrectos, ya sea dentro de la base de datos o fuera de ella.

El capítulo 14 trata la recuperación y el 15 la concurrencia. *Nota:* Partes de estos capítulos aparecieron originalmente, aunque en forma ligeramente diferente, en el libro *An Introduction to Database Systems: Volume II* (Addison-Wesley, 1983).

CAPITULO **14**

Recuperación

14.1 INTRODUCCIÓN

Como mencioné al inicio de esta parte del libro, los temas de este capítulo y el siguiente (recuperación y concurrencia) están muy interrelacionados, ya que ambos son parte del tema más general de la *administración de transacciones.* Pero para simplificar su explicación, es preferible mantenerlos por separado en la medida de lo posible (al menos hasta que hayamos acabado de describir algunos de los conceptos básicos). Por lo tanto, el presente capítulo está enfocado principalmente en la recuperación, y dejamos la concurrencia para el capítulo 15. Sin embargo, de vez en cuando aparecerán en este capítulo algunas referencias inevitables a la concurrencia.

La **recuperación** en un sistema de base de datos significa principalmente la recuperación de la propia base de datos; es decir, el restablecimiento de la misma a un estado correcto (o mejor dicho, consistente*) después de que alguna falla haya ocasionado que el estado actual sea inconsistente, o al menos eso parezca. Los principios en los que está basada la recuperación son bastante simples y pueden ser resumidos en una palabra: **redundancia.** (Redundancia al nivel físico, yaque por las razones explicadas a fondo en la parte III de este libro, por lo general no es necesario que tal redundancia aparezca al nivel lógico.) En otras palabras, la forma de asegurar que la base de datos sea recuperable, es garantizando que cualquier parte de la información que contiene puede ser reconstruida a partir de otra información guardada redundantemente en algún otro lugar del sistema.

Antes de continuar, debemos dejar claro que las ideas sobre la recuperación —de hecho, las ideas sobre el procesamiento de transacciones en general— son en cierta forma independientes del hecho que el sistema subyacente sea relacional o de cualquier otro tipo. (Por otro lado, también debemos decir que la mayor parte del trabajo teórico sobre el procesamiento de transacciones se ha hecho, y continúa haciéndose, en un contexto relacional.) También debemos dejar claro que éste es un tema enorme y que todo lo que podemos esperar aquí es presentar algunas de las ideas más importantes y básicas. Vea la sección "Referencias y bibliografía", en especial la referencia [14.12], para algunas sugerencias de lecturas adicionales, así como los ejercicios y respuestas para una breve explicación de temas adicionales.

El plan del capítulo es el siguiente. Después de esta breve introducción, las secciones 14.2 y 14.3 explican la noción fundamental de una *transacción* y la idea relacionada de la *recuperación de transacciones* (es decir, la recuperación de la base de datos después de que por alguna razón ha

---

**Consistente* aquí significa "satisfacer todas las restricciones de integridad conocidas". Por lo tanto, observe que *consistente* no necesariamente significa *correcto;* ya que un estado correcto necesariamente debe ser consistente, pero un estado consistente puede seguir siendo incorrecto en el sentido de que no refleja con precisión el verdadero estado de las cosas en el mundo real. "Consistente" puede ser definido como "correcto en lo que se refiere al sistema".

fallado una transacción individual). La sección 14.4 continúa expandiendo las ideas anteriores en el ámbito más amplio de la recuperación del *sistema* (es decir, la recuperación después de que un error del sistema ha ocasionado que todas las transacciones actuales fallen simultáneamente). La sección 14.5 se desvía un poco hacia la recuperación del *medio* (es decir, la recuperación después de que la base de datos ha sufrido algún tipo de daño físico; por ejemplo, por un roce de las cabezas con el disco). La sección 14.6 presenta el concepto extremadamente importante de la *confirmación en dos fases.* La sección 14.7 describe las características relevantes de SQL. Por último, la sección 14.8 presenta un resumen y algunos comentarios finales.

Una última nota preliminar: A lo largo de este capítulo damos por hecho que estamos en un ambiente de base de datos "grande" (compartido y multiusuario). Los DBMSs (Sistemas de Administración de Bases de Datos) "pequeños" (no compartidos o de un solo usuario) proporcionan generalmente muy poco o ningún soporte para la recuperación; en su lugar, la recuperación es vista como responsabilidad del usuario (lo que implica que el usuario debe realizar copias de seguridad periódicas de la base de datos y rehacer el trabajo manualmente cuando ocurra una falla).

14.2 TRANSACCIONES

Como indiqué en la sección **14.1,** comenzamos nuestras explicaciones analizando la noción fundamental de una **transacción.** Una transacción es una **unidad de trabajo lógica.** Considere el siguiente ejemplo. Suponga que la varrel de partes P incluye un atributo adicional CANTOT, que representa el total de la cantidad enviada para la parte en cuestión; en otras palabras, el valor de CANTOT para cualquier parte deberá ser igual a la suma de todos los valores CANT tomados de todos los envíos de esa parte (en la terminología del capítulo 8, ésta es una *restricción de la base de datos).* Ahora considere el procedimiento de seudocódigo mostrado en la figura 14.1, cuya finalidad es agregar a la base de datos un nuevo envío para el proveedor V5 y la parte Pl,

```
    BEGIN TRANSACTION ;

    INSERT INTO VP RELATION {       V#  ( ' S5' ), P#
    TUPLE { V# P# CANT IF ocurre    ( ■P1' ), CANT (
    algún error THEN GO             1000 ) } } ; TO UNDO
                                    ; END IF  ;


    UPDATE P WHERE P# = P# ('P1')      (  1000  ) ;
    CANTOT :■ CANTOT + CANT IF         UNDO  ;  END IF  ;
    ocurre algún error THEN GO TO

    COMMIT ; GO TO
    FINISH  ;

UNDO  :
ROLLBACK  ;

FINISH  :
RETURN  ;
```

Figura 14.1 Una transacción de ejemplo (seudocódigo).

con una cantidad de 1000, (la instrucción INSERT inserta el nuevo envío, UPDATE actualiza el valor de CANTOT para la parte Pl en concordancia).

El punto a destacar en el ejemplo es que lo que supone ser una sola operación atómica, "añadir un nuevo envío", implica de hecho *dos* actualizaciones a la base de datos: una operación INSERT y una operación UPDATE. Lo que es más, la base de datos ni siquiera es constante entre esas dos actualizaciones, ya que viola temporalmente la restricción que indica que el valor de CANTOT para la parte P1 debe ser igual ala suma de todos los valores CANT para esa parte. Por lo tanto, una unidad de trabajo lógica (por ejemplo, una transacción) no es necesariamente una sola operación de la base de datos, sino que en general es una *secuencia* de varias de estas operaciones que transforman un estado consistente de la base de datos en otro estado consistente sin que sea necesario conservar la consistencia en todos los puntos intermedios.

Ahora queda claro que lo que no está permitido en el ejemplo es que una de las actualizaciones sea ejecutada y la otra no, ya que esto dejaría a la base de datos en un estado inconsistente. Claro que idealmente quisiéramos una garantía absoluta de que ambas actualizaciones serán ejecutadas. Desgraciadamente, es imposible ofrecer esta garantía, ya que siempre existe la posibilidad de que algo salga mal y de que suceda en el peor momento posible. Por ejemplo, podría ocurrir una caída del sistema entre INSERT y UPDATE, o un desborde aritmético en UPDATE, entre otras cosas.* Pero un sistema que soporta la **administración de transacciones** proporciona lo mejor que hay para dar tal garantía. Específicamente garantiza que si la transacción ejecuta algunas actualizaciones y luego, por cualquier razón, ocurre una falla antes de que la transacción alcance su terminación planeada, entonces *esas actualizaciones serán deshechas.* Por lo tanto, la transacción *o* se ejecuta *o* se cancela totalmente (es decir, como si nunca hubiera sido ejecutada). De esta forma, es posible hacer que una secuencia de operaciones que básicamente no es atómica parezca como si lo fuera, desde un punto de vista externo.

El componente del sistema que proporciona esta atomicidad, o semblanza de atomicidad, es conocido como **administrador de transacciones** (también como **monitor de procesamiento de transacciones** o **monitor PT)** y las operaciones COMMIT y ROLLBACK son la clave de cómo funciona:

- La operación **COMMIT** indica la finalización de una transacción *satisfactoria:* indica al administrador de transacciones que una unidad de trabajo lógica ha concluido satisfactoria mente, que la base de datos está o debería estar nuevamente en un estado consistente y que todas las actualizaciones efectuadas por esa unidad de trabajo ahora pueden ser "confirma das" o definitivas.
- Por el contrario, la operación **ROLLBACK** indica la finalización de una transacción *no sa tisfactoria:* indica al administrador de transacciones que algo ha salido mal, que la base de datos puede estar en un estado inconsistente y que todas las actualizaciones realizadas hasta este momento por la unidad de trabajo lógica deben ser "revertidas" o deshechas.

Por lo tanto, en el ejemplo emitimos una instrucción COMMIT cuando las dos actualizaciones se llevan a cabo satisfactoriamente. Esto confirmará los cambios en la base de datos y los hará permanentes. Sin embargo, si algo sale mal —es decir, si cualquiera de las actualizaciones indica una condición de error— emitimos ROLLBACK para deshacer cualquier cambio realizado

---

*A una *caída* de sistema también se le conoce como falla *global* o de *sistema;* y a la falla de un programa individual, como el caso de un desbordamiento, también se le conoce como falla *local.* Vea las secciones 14.3 y 14.4.

hasta ahora. *Nota:* Aun si emitimos COMMIT en lugar de ROLLBACK, el sistema debería verificar la integridad de la base de datos, detectar cuándo la base es inconsistente y forzar una instrucción ROLLBACK. Sin embargo, en realidad no podemos dar por hecho que el sistema está consciente de todas las restricciones pertinentes, por lo que es necesario que el usuario emita el ROLLBACK. Hasta el momento de la publicación de este libro, los DBMS comerciales no verificaban en gran medida la integridad usando COMMIT.

Dicho sea de paso, debemos remarcar que una aplicación real no sólo actualizará la base de datos (o tratará de hacerlo), sino que también enviará algún tipo de mensaje al usuario final indicándole lo que ha sucedido. En el ejemplo, podríamos enviar el mensaje "Se añadió el envío" si la instrucción COMMIT es alcanzada, o bien "Error, no se añadió el envío", en el caso contrario. El manejo de mensajes tiene a su vez implicaciones adicionales para la recuperación. Para una explicación más amplia vea la referencia [14.12].

Nota: En este momento tal vez se pregunte cómo es posible deshacer una actualización. La respuesta, por supuesto, es que el sistema mantiene una **bitácora** o **diario** en cinta o (más comúnmente) en disco, y ahí guarda los detalles de todas las actualizaciones, en particular las imágenes anterior y posterior del objeto actualizado. Por lo tanto, si necesita deshacer una actualización en particular, el sistema puede usar la anotación (o registro) correspondiente de la bitácora para restaurar el objeto actualizado a su valor anterior.

(De hecho, el párrafo anterior está excesivamente simplificado. En la práctica, la bitácora consta de dos partes: una *activa* o en línea y otra *archivada* o fuera de línea. La parte en línea es la que se usa durante la operación normal del sistema para grabar los detalles de las actualizaciones conforme son realizadas, y normalmente es guardada en disco. Cuando la parte en línea se llena, su contenido es transferido a la parte fuera de línea, la cual, gracias a que siempre tiene un procesamiento secuencial, puede guardarse en cinta.)

Otro punto importante: el sistema debe garantizar que las instrucciones individuales sean atómicas por sí mismas (todo o nada). Esta consideración llega a ser particularmente significativa en un sistema relational, en donde las instrucciones son en el nivel de conjunto y por lo general operan en muchas tupias a la vez; no debe ser posible que una instrucción de éstas falle a la mitad y deje a la base de datos en un estado inconsistente (por ejemplo, con algunas tupias actualizadas y otras no). En otras palabras, si ocurre un error a la mitad de una instrucción de éstas, entonces la base de datos deberá permanecer sin ningún cambio. Además, como expliqué en los capítulos 8 y 9, lo mismo ocurre aun cuando la instrucción provoca que se realicen operaciones adicionales ocultas, tal como sucede (por ejemplo) en una regla DELETE de clave externa que especifica una acción referencial CASCADE.

RECUPERACIÓN DE TRANSACCIONES

Una transacción comienza con la ejecución satisfactoria de una instrucción BEGIN TRANSACTION y termina con la ejecución satisfactoria de una instrucción COMMIT o ROLLBACK. COMMIT establece lo que es conocido, entre muchas otras acepciones, como **punto de confirmación** (también se conoce como **punto de sincronización,** especialmente en productos comerciales). Un punto de confirmación corresponde entonces al final de una unidad de trabajo lógica y por lo tanto, a un punto en el cual la base de datos está o debería estar en un estado consistente. Por el contrario, ROLLBACK regresa la base de datos al estado en que estaba antes de BEGIN TRANSACTION, lo que en efecto significa regresar al punto de confirmación anterior. (La frase "punto de confirmación anterior" sigue siendo adecuada, aun en el caso de que sea la

primera transacción en el programa; si coincidimos en que la primera BEGIN TRANSACTION del programa está estableciendo tácitamente un "punto de confirmación" inicial.)

Nota: A lo largo de esta sección, el término "base de datos" significa en realidad sólo aquella parte de la base de datos a la que la transacción en consideración está teniendo acceso. Es posible que otras transacciones estén en ejecución paralela y hagan cambios a sus propias partes, por lo que también es posible que la "base de datos total" *no* esté en un estado completamente consistente en un punto de confirmación. Sin embargo, como expliqué en la sección 14.1, en este capítulo ignoraremos en la medida de lo posible la posibilidad de transacciones concurrentes. Por supuesto, esta simplificación en realidad no afecta el tema que tratamos.

Al establecer un punto de confirmación:

1. Todas las actualizaciones hechas por el programa en ejecución desde el punto de confirma ción anterior, son confirmadas; es decir, se **vuelven permanentes.** Antes del punto de confir mación, todas estas actualizaciones deben ser vistas como *meramente tentativas,* en el sentido de que pueden ser deshechas posteriormente. Es un hecho que una vez que una transacción ha sido confirmada nunca podrá ser deshecha (ésta es la definición de "confirmada").
2. Todo el posicionamiento de la base de datos se pierde y todos los bloqueos de tupias son liberados. El "posicionamiento de la base de datos" se refiere en este caso a la idea de que en cualquier momento, un programa en ejecución tendrá una direccionabilidad hacia cier tas tupias (por ejemplo, mediante determinados *cursores* en el caso de SQL, como expliqué en el capítulo 4); esta direccionabilidad se pierde en un punto de confirmación. Los "blo queos de tupias" son explicados en el siguiente capítulo. *Nota:* Algunos sistemas propor cionan una opción por la cual el programa puede conservar la direccionabilidad hacia determinadas tupias (y por lo tanto, conservar determinados bloqueos de tupias) entre una transacción y la siguiente. Para una explicación más amplia vea la sección 14.7.

El párrafo 2 anterior —salvo el comentario acerca de la posible retención de alguna direccionabilidad, y por tanto, la posible conservación de determinados bloqueos de tupias— también se aplica cuando la transacción termina con ROLLBACK en vez de COMMIT. El párrafo 1, por supuesto, no presenta el mismo caso.

Observe cuidadosamente que COMMIT y ROLLBACK terminan la *transacción* y no el programa. Por lo general, una sola ejecución del programa consta de una *secuencia* de varias transacciones ejecutándose una tras otra, como ilustra la figura 14.2.

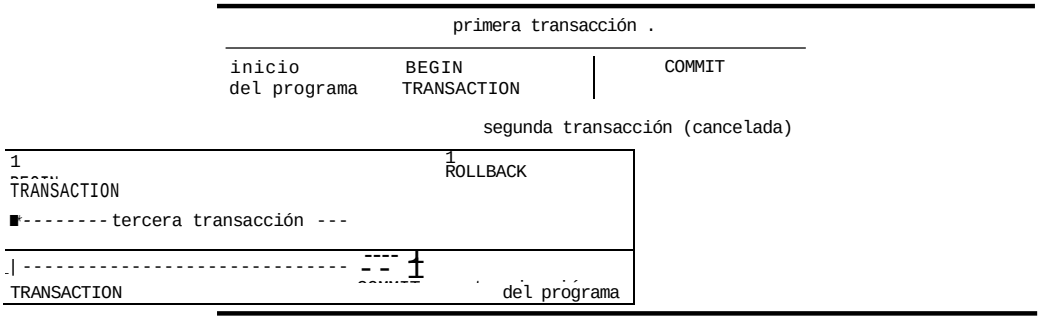


Figura 14.2 La ejecución del programa es una secuencia de transacciones.

Regresemos ahora al ejemplo de la sección anterior (figura 14.1). En ese ejemplo incluimos pruebas explícitas para errores y emitimos una instrucción ROLLBACK explícita cuando fue detectado algún error. Pero el sistema por supuesto no puede dar por hecho que los programas de aplicación siempre incluirán pruebas explícitas para todos los errores posibles. Por lo tanto, el sistema emitirá una instrucción ROLLBACK *implícita* para cualquier transacción que falle por cualquier razón, para llegar a su terminación planeada (donde "terminación planeada" significa una instrucción COMMIT o ROLLBACK explícitas).

Por lo tanto, ahora podemos ver que las transacciones no sólo representan la unidad de trabajo sino también la unidad de **recuperación.** Si una transacción es confirmada satisfactoriamente, el sistema garantizará que sus actualizaciones queden grabadas permanentemente en la base de datos, aunque el sistema llegue a fallar en el siguiente momento. Por ejemplo, es probable que el sistema falle después de haber recibido a COMMIT, pero antes de escribir físicamente las actualizaciones en la base de datos; éstas podrían seguir esperando en un búfer de memoria principal y por lo tanto, estar perdidas en el momento de la falla. Aunque esto suceda, el procedimiento de reinicio del sistema grabará esas actualizaciones en la base de datos. Éste es capaz de descubrir los valores que hay que escribir, examinando las entradas relevantes de la bitácora. (De aquí se deduce que la bitácora debe ser escrita físicamente antes de que termine el procesamiento de COMMIT; lo que se conoce como **regla de escritura anticipada de la bitácora.)** Por lo tanto, el procedimiento de reinicio recuperará cualquier transacción que haya terminado satisfactoriamente pero que no haya podido lograr que sus actualizaciones fueran escritas físicamente antes de la caída. Por lo tanto, como dije anteriormente, las transacciones son la unidad de recuperación. *Nota:* En el siguiente capítulo veremos que también son la unidad de *concurrencia.* Además, puesto que supuestamente transforman un estado consistente hacia otro estado consistente de la base de datos, también pueden ser vistas como una unidad de *integridad.* Vea el capítulo 8.

Las propiedades ACID

En conformidad con la referencia [14.14], podemos resumir esta sección y la anterior diciendo que las transacciones tienen cuatro propiedades importantes: *atomicidad, consistencia, aislamiento* y *durabilidad* (las que se conocen coloquialmente como "propiedades ACID").

- **Atomicidad.** Las transacciones son atómicas (todo o nada).
- **Consistencia.** Las transacciones conservan la consistencia de la base de datos. Es decir, una transacción transforma un estado consistente de la base de datos en otro igual, sin necesi dad de conservar la consistencia en todos los puntos intermedios.
- **Aislamiento.** Las transacciones están aisladas entre sí. Es decir, aunque en general hay mu chas transacciones ejecutándose en forma concurrente, las actualizaciones de una transacción dada están ocultas ante las demás, hasta que esa transacción sea confirmada. Otra forma de decir lo mismo es que para dos transacciones distintas, como 77 y 72, 77 podrá ver las actualizaciones de 72 (después de que 72 haya sido confirmada) o 72 podrá ver las de 77 (después de que *TI* haya sido confirmada), pero no podrán suceder ambas cosas. Para una explicación más amplia vea el capítulo 15.
- **Durabilidad.** Una vez que una transacción es confirmada, sus actualizaciones sobreviven en la base de datos aun cuando haya una caída posterior del sistema.

14.4 RECUPERACIÓN DEL SISTEMA

El sistema debe estar preparado para recuperarse, no solamente de las fallas puramente locales como una condición de desborde dentro de una transacción individual, sino también de las fallas "globales", como una falla en el suministro eléctrico. Por definición, una falla local afecta solamente a la transacción en la cual está ocurriendo; tales fallas ya las he tratado en las secciones 14.2 y 14.3. Por el contrario, una falla global afecta a todas las transacciones que están en progreso en el momento de la misma y por lo tanto, tiene implicaciones significativas al nivel del sistema. En esta sección y la siguiente, consideraremos brevemente lo que implica la recuperación de una falla global. Tales fallas caen en dos categorías amplias:

- Las **fallas del sistema** (por ejemplo, falla en el suministro eléctrico) que afectan a todas las transacciones que están actualmente en progreso, pero que no dañan físicamente a la base de datos. A la falla de sistema se le conoce a veces como una *caída blanda.*
- Las **fallas del medio** (por ejemplo, un roce de las cabezas con el disco) que causan daño a la base de datos o a alguna parte de ella, y afectan al menos las transacciones que están usan do actualmente esa parte. A una falla del medio se le conoce a veces como una *caída dura.*

A continuación tratamos las fallas del sistema; las fallas del medio son tratadas en la sección 14.5.

El punto central con relación a la falla del sistema es que se *pierde el contenido de la memoria principal* (en particular se pierden los búferes de la base de datos). Ya no es posible conocer el estado preciso de cualquier transacción que estaba en progreso al momento de la falla y por lo tanto, tal transacción nunca podrá terminar satisfactoriamente y deberá *ser deshecha* cuando el sistema vuelva a iniciar.

Además, tal vez también sea necesario al momento de reinicio (como sugerí en la sección 14.3) *volver a realizar* determinadas transacciones que se completaron satisfactoriamente antes de la falla pero que no pudieron lograr que sus actualizaciones fueran transferidas desde los búferes de la base de datos hacia la base de datos física.

Por lo tanto, se presenta la pregunta obvia: ¿Cómo sabe el sistema cuáles transacciones debe deshacer y cuáles rehacer al momento del reinicio? La respuesta es la siguiente. A determinados intervalos prescritos, por lo general cuando ya se ha escrito una determinada cantidad de entradas en la bitácora, el sistema automáticamente **hace un punto de verificación.** Hacer un punto de verificación involucra (a) escribir físicamente ("escritura forzada") el contenido de los búferes de la base de datos hacia la base de datos física y (b) escribir físicamente un **registro de punto de verificación** especial en la bitácora física. El registro de punto de verificación da una lista de todas las transacciones que estaban en progreso en el momento en que se realizó el punto de verificación. Para ver la manera en que se usa esta información, considere la figura 14.3, que se lee de la siguiente forma (tome en cuenta que en la figura el tiempo fluye de izquierda a derecha):

- En el tiempo *tf* sucede una falla de sistema.
- El punto de verificación más reciente anterior al tiempo *tf,* se tomó en el tiempo *te.*
- Las transacciones de tipo *TI* terminaron (satisfactoriamente) antes del tiempo *te.*
- Las transacciones de tipo *T2* se iniciaron antes del tiempo *te,* y terminaron (satisfactoria mente) después del tiempo *te* y antes del tiempo *tf.*
- Las transacciones de tipo *T3* también se iniciaron antes del tiempo *te,* pero no habían ter minado en el tiempo *tf.*
- Las transacciones de tipo *T4* se iniciaron después del tiempo *te* y terminaron (satisfactori amente) antes del tiempo *tf.*
- Por último, las transacciones de tipo 75 también se iniciaron después del tiempo *te,* pero no habían terminado en el tiempo *tf.*

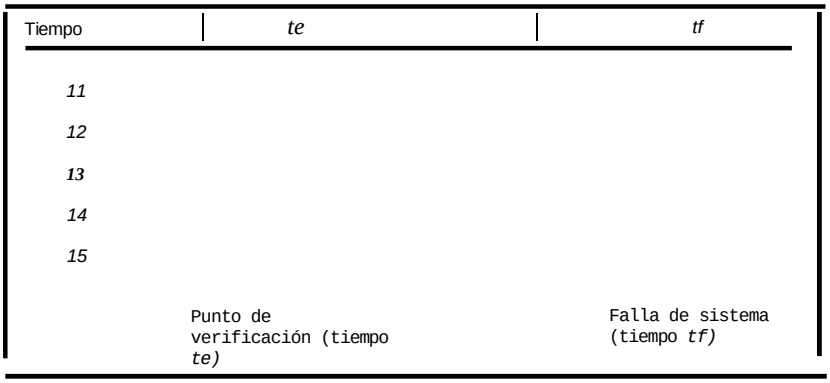


Figura 14.3 Cinco categorías de transacciones.

Debe quedar claro que cuando se vuelve a iniciar el sistema, las transacciones de tipo *T3 y T5* deben deshacerse, y las de tipo *T2yT4* deben rehacerse. Sin embargo, observe que las transacciones de tipo *TI* no entran en el proceso de reinicio, ya que sus actualizaciones fueron forzadas hacia la base de datos en el tiempo *te* como parte del proceso del punto de verificación. Observe también que las transacciones que terminaron en forma no satisfactoria (es decir con una instrucción deshacer) antes del tiempo *tf* tampoco entran en el proceso de reinicio (¿por qué no?).

Por lo tanto, al reiniciar, el sistema realiza el siguiente procedimiento para identificar todas las transacciones de tipos *TI* a *T5:*

1. Comienza con dos listas de transacciones, la lista DESHACER y la lista REHACER. Iguala la lista DESHACER con la lista de todas las transacciones dadas en el registro de punto de verificación más reciente y deja vacía la lista REHACER.
2. Busca hacia delante en la bitácora comenzando en el registro del punto de verificación.
3. Si encuentra una entrada BEGIN TRANSACTION en la bitácora para la transacción *T,* añade *T* a la lista DESHACER.
4. Si encuentra una entrada COMMIT para la transacción *T,* mueve a *T* de la lista DESHACER a la lista REHACER.
5. Cuando llega al fin de la bitácora, las listas DESHACER y REHACER identifican a las tran sacciones de tipo *T3 y T5,* así como a las de tipo *T2* y *T4,* respectivamente.

Ahora el sistema trabaja hacia atrás en la bitácora, deshaciendo las transacciones que están en la lista DESHACER, y luego vuelve a trabajar hacia delante volviendo a hacer las transacciones que están en la lista REHACER.* *Nota:* A la restauración de la base de datos a un estado consistente deshaciendo el trabajo, en ocasiones se le llama *recuperación hacia atrás.* En forma similar, a la restauración a un estado consistente rehaciendo el trabajo, se le llama *recuperación hacia adelante.*

*Notará que nuestra descripción del proceso de recuperación del sistema está muy simplificada. En particular, muestra al sistema realizando primero operaciones de "deshacer" y luego operaciones de "rehacer". Los primeros sistemas funcionaban de esa forma, pero por razones de eficiencia los sistemas modernos generalmente hacen las cosas al revés (vea por ejemplo las referencias [4.17] y [4.19]).

Por último, cuando ha terminado toda la actividad de recuperación, es entonces (y sólo entonces) que el sistema está listo para aceptar más trabajo.

14.5 RECUPERACIÓN DEL MEDIO

Nota: El tema de recuperación del medio es un poco diferente al de recuperación de transacciones y del sistema. Lo incluimos aquí únicamente para ampliar la explicación.

Para repetir lo que dije en la sección 14.4, una falla del medio podría ser un roce de cabezas o una falla del controlador de disco, en la cual parte de la base de datos se destruye físicamente. En general, la recuperación de este tipo de fallas implica básicamente volver a cargar (o *restaurar)* la base de datos a partir de una copia de seguridad (o *respaldo),* y luego usar la bitácora —tanto las partes activas como las archivadas— para volver a hacer todas las transacciones que se terminaron a partir de que se realizó el respaldo. No hay necesidad de deshacer transacciones que estaban en progreso al momento de la falla, ya que por definición todas las actualizaciones de esas transacciones ya han sido "deshechas" (de hecho se perdieron).

La necesidad de poder realizar la recuperación del medio implica el uso de una *utilería de vaciado y restauración* (o *descarga/recarga).* La parte de vaciado de esa utilería se usa para hacer respaldos de la base de datos cuando se requiera. (Tales respaldos pueden ser guardados en cinta u en otro medio de almacenamiento, ya que no es necesario que estén en un medio de acceso directo.) Después de que ha habido una falla en el medio, se usa la parte de restauración de la utilería para volver a crear la base de datos a partir de un respaldo específico.

14.6 CONFIRMACIÓN DE DOS FASES

En esta sección trataremos brevemente un desarrollo muy importante del concepto básico de confirmar/deshacer llamado confirmación de **dos** fases. La confirmación de dos fases es importante siempre que una transacción dada pueda interactuar con varios "administradores de recursos" independientes, donde cada uno administra su propio conjunto de recursos recuperables y mantiene su propia bitácora de recuperación.* Por ejemplo, considere una transacción que esté ejecutándose en una mainframe IBM que actualiza una base de datos IMS y otra DB2 (dicho sea de paso, tal transacción es perfectamente válida). Si la transacción termina satisfactoriamente se deben confirmar *todas* sus actualizaciones, tanto para los datos de IMS como para los de DB2; por el contrario, si falla, *todas* sus actualizaciones deben ser deshechas. En otras palabras, no debe ser posible confirmar las actualizaciones IMS y cancelar las actualizaciones DB2, o *viceversa,* ya que entonces la transacción no sería atómica.

Esto nos lleva a concluir que no tiene sentido que la transacción emita, digamos, un COMMIT para IMS y un ROLLBACK para DB2; y a que incluso si emitiera la misma instrucción para ambos, el sistema aún podría fallar entre una y otra, con resultados desafortunados. Por lo

---

*En particular, esto es importante en el contexto de los sistemas de bases de datos distribuidas; por esa razón, trato el tema con mayor detalle en el capítulo 20.

tanto, en vez de ello, la transacción emite un solo COMMIT (o ROLLBACK) **a nivel sistema.** Ese COMMIT o ROLLBACK "global" es manejado por un componente del sistema llamado **coordinador,** cuya tarea es garantizar que ambos administradores de recursos (es decir, IMS y DB2, en este ejemplo) confirmen o deshagan *al unísono* las actualizaciones de las que son responsables, y además proporcionar esa garantía *aunque el sistema falle a mitad del proceso.* Y es el protocolo de confirmación de dos fases el que permite que el coordinador proporcione esta garantía.

Funciona de la siguiente forma. Por simplicidad, supongamos que la transacción ha terminado satisfactoriamente su procesamiento de base de datos y por lo tanto, la instrucción al nivel sistema que emite es COMMIT y no ROLLBACK. Al recibir la petición de COMMIT, el coordinador realiza el siguiente proceso de dos fases:

1. Primero, da instrucciones a todos los administradores de recursos a fin de que estén listos para manejar la transacción "de una u otra forma". En la práctica esto significa que cada **participante** en el proceso (es decir, cada administrador de recursos involucrado) debe forzar todos los registros de bitácora de los recursos locales usados por la transacción, hacia su propia bitácora física (es decir, hacia el almacenamiento no volátil); esto con el fin de que, sin importar qué pase después, el administrador de recursos tenga ahora un *registro permanente* del trabajo que hizo a nombre de la transacción y por lo tanto sea capaz de con firmar o deshacer las actualizaciones según sea necesario. Suponiendo que la escritura forzada es satisfactoria, el administrador de recursos responde ahora un "OK" al coordina dor, y en caso contrario responde "No OK".
2. Cuando el coordinador ha recibido las respuestas de todos los participantes, fuerza una en trada en su propia bitácora física registrando su decisión con respecto a la transacción. Si todas las respuestas fueron "OK", esa decisión es "confirmar", y si alguna respuesta fue "No OK", la decisión es "deshacer". De cualquier forma, el coordinador informa después su de cisión a cada participante y luego *cada participante debe confirmar o deshacer la transac ción ¡ocalmente según se le indica.* Observe que cada participante debe hacer lo que dice el coordinador en la fase 2; éste es el protocolo. Observe también que la apariencia del regis tro de decisión en la bitácora física del coordinador es lo que marca la transición de la fase 1 a la 2.

Ahora, si el sistema falla durante el proceso general, el procedimiento de reinicio buscará el registro de decisión en la bitácora del coordinador. Si lo encuentra, el proceso de confirmación de dos fases puede continuar donde se quedó. Si no lo encuentra, da por hecho que la decisión fue "deshacer" y de nuevo el proceso puede terminar en forma adecuada. *Nota:* Vale la pena señalar que si el coordinador y los participantes están ejecutándose en máquinas diferentes, como puede suceder en un sistema distribuido (vea el capítulo 20), una falla en la parte del coordinador puede mantener a algún participante esperando mucho tiempo a que llegue la decisión del coordinador y mientras *está* esperando, cualquier actualización hecha por la transacción a través de ese participante, deberá mantenerse oculta ante otras transacciones (es decir, probablemente esas actualizaciones tendrán que permanecer *bloqueadas,* como menciono en el siguiente capítulo).

Insistimos en que el administrador de comunicaciones de datos (o el administrador CD; vea el capítulo 2) también puede ser visto como un administrador de recursos en el sentido que aquí mencionamos. Es decir, los mensajes también pueden ser vistos como un recurso recuperable (similar a las bases de datos) y el administrador CD debe poder participar en el proceso de

confirmación de dos fases. Para una explicación adicional de este punto, y de la idea completa de la confirmación de dos fases en general, vea la referencia [14.12],

14.7 PROPIEDADES DE SQL

El soporte de SQL para las transacciones y por lo tanto, para la recuperación basada en transacciones, sigue los lincamientos generales descritos en las secciones anteriores. En particular, SQL soporta las instrucciones **COMMIT y ROLLBACK** usuales (en ambos casos, con una palabra clave adicional opcional WORK, como vimos en el capítulo 4). Estas instrucciones fuerzan un CLOSE para cada cursor abierto, ocasionando que se pierda todo el posicionamiento en la base de datos. *Nota:* Algunas implementaciones de SQL proporcionan una forma para impedir este CLOSE automático y la pérdida del posicionamiento para COMMIT (pero no para ROLLBACK). Por ejemplo, DB2 soporta una opción WITH HOLD en una declaración de cursor; COMMIT no cierra este cursor sino que lo deja abierto y posicionado de tal forma que el siguiente FETCH lo moverá hacia el próximo registro en la secuencia. Por lo tanto, ya no es necesario el complejo código de reposicionamiento que pudiera haberse necesitado en el siguiente OPEN. Esta característica está incluida actualmente en SQL3 (vea el apéndice B).

Una diferencia entre el soporte de SQL para las transacciones y los conceptos generales que describo en este capítulo es que SQL no incluye ninguna instrucción BEGIN TRANSACTION explícita. En vez de ello, cada vez que el programa ejecuta una instrucción de **"inicio de transacción"** y ya no tiene una transacción en progreso, inicia implícitamente una transacción. (Así como sucede con la opción de "conservación de cursor" tratada anteriormente, es probable que en alguna versión futura se añada una instrucción BEGIN TRANSACTION explícita; esta instrucción está incluida actualmente en SQL3.) Los detalles de qué instrucciones SQL representan un "inicio de transacción" están más allá del alcance de este libro; aunque basta decir que todas las instrucciones ejecutables que se han tratado en los capítulos anteriores *representan* un inicio de transacción, en cambio, COMMIT y ROLLBACK es obvio que no lo representan. Una instrucción especial, llamada **SET TRANSACTION,** se usa para definir determinadas características de la siguiente transacción a iniciar (sólo es posible ejecutar SET TRANSACTION cuando no hay transacciones en progreso y cuando no es por sí misma un inicio de transacción). Aquí sólo tratamos dos de estas características que son el *modo de acceso* y el *nivel de aislamiento.* La sintaxis es la siguiente:

```
SET TRANSACTION <lista de opciones separadas con comas>;
```

en donde *<lista de opciones separadas con comas>* especifica un modo de acceso, un nivel de aislamiento o ambos.

- El **modo de acceso** es READ ONLY o READ WRITE. En caso de no especificar ninguno, se da por hecho a READ WRITE; a menos que esté especificado el nivel de aislamiento READ UNCOMMITTED, en cuyo caso se da por hecho a READ ONLY. Si se especifi ca READ WRITE, el nivel de aislamiento no debe ser READ UNCOMMITTED.
- El **nivel de aislamiento** toma la forma de ISOLATION LEVEL *<aislamiento>*, donde *<aislamiento>* es READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ o SERIALIZABLE. Para una explicación adicional vea el capítulo 15.

14.8 RESUMEN

En este capítulo hemos presentado una introducción necesariamente corta del tema de **administración de transacciones.** Una transacción es una **unidad de trabajo lógica** y también una **unidad de recuperación** (además de una unidad de concurrencia y una unidad de integridad, vea los capítulos 15 y 8, respectivamente). Las transacciones poseen las **propiedades ACID** de **atomicidad, consistencia, aislamiento y durabilidad. La administración de transacciones** es la tarea de supervisar la ejecución de transacciones, en forma tal que se pueda garantizar que poseen estas propiedades importantes. De hecho, el propósito general del sistema podría ser bien definido como la **ejecución confiable de transacciones.**

Las transacciones se inician con **BEGIN TRANSACTION** y terminan ya sea con **COMMIT** (terminación *satisfactoria)* o con **ROLLBACK** (terminación *no satisfactoria).* COMMIT establece un **punto de confirmación** (las actualizaciones se vuelven permanentes) y ROLLBACK regresa la base de datos al punto de confirmación anterior (las actualizaciones se deshacen). Si una transacción no llega a su terminación planeada el sistema *fuerza* un ROLLBACK **(recuperación de transacción).** Para poder deshacer (o rehacer) actualizaciones el sistema mantiene una **bitácora** de recuperación. Además, los registros de bitácora para una transacción dada deben ser escritos en la bitácora física antes de terminar el procesamiento del COMMIT para esa transacción (la **regla de escritura anticipada de la bitácora).**

El sistema también garantiza las propiedades ACID de las transacciones ante una caída del sistema. Para brindar esta garantía, el sistema debe (a) **rehacer** todo el trabajo realizado por las transacciones que terminaron satisfactoriamente antes de la caída y (b) **deshacer** todo el trabajo realizado por las transacciones que se iniciaron pero no terminaron antes de la caída. Esta actividad de **recuperación del sistema** es realizada como parte del procedimiento de **reinicio** del sistema (conocido a veces como el procedimiento de *reinicio/recuperación).* El sistema descubre qué trabajo hay que rehacer y cuál hay que deshacer, examinando el **registro de punto de verificación** más reciente. Los registros de punto de verificación se escriben en la bitácora a intervalos preestablecidos.

El sistema también proporciona la **recuperación del medio** al restaurar la base de datos a partir de un **vaciado** previo y después, usando la bitácora, rehacer el trabajo que se terminó desde que se hizo el vaciado. Se necesitan las **utilerías** de vaciado/restauración para soportar la recuperación del medio.

Los sistemas que permiten que las transacciones interactúen con dos o más **administradores de recursos** (por ejemplo, dos DBMSs diferentes, o un DBMS y un administrador CD), deben usar un protocolo llamado **confirmación de dos fases** si es que van a mantener la propiedad de atomicidad de la transacción. Las dos fases son: (a) la fase de **preparación,** en la cual el **coordinador** da instrucciones a todos los **participantes** para que "estén listos para actuar de una u otra forma", y (b) la fase de **confirmación,** en la cual el coordinador da instrucciones a todos los participantes, suponiendo que respondieron satisfactoriamente durante la fase de preparación, para que realicen la confirmación actual (o en caso contrario, den la instrucción deshacer).

Con relación al soporte para la recuperación en SQL, éste proporciona instrucciones **COMMIT** y **ROLLBACK** explícitas (pero no la instrucción BEGIN TRANSACTION explícita). También soporta la instrucción **SET TRANSACTION,** la cual permite que el usuario especifique el **modo de acceso y** el **nivel de aislamiento** de la siguiente transacción a iniciar.

Un último punto: a lo largo de este capítulo hemos estado suponiendo tácitamente un ambiente de programación de aplicaciones. Sin embargo, todos los conceptos que se han tratado son

aplicables también al ambiente de usuario final (aunque pueden estar un poco más ocultos a este nivel). Por ejemplo, los productos SQL permiten normalmente que el usuario utilice instrucciones SQL en forma interactiva desde una terminal. En general, cada una de estas instrucciones SQL interactivas es tratada como una transacción por derecho propio; es común que el sistema emita un COMMIT automático a nombre del usuario después de haber ejecutado la instrucción SQL (o por supuesto, un ROLLBACK automático en caso de falla). Sin embargo, algunos sistemas permiten que el usuario inhiba estos COMMITS automáticos y ejecute en su lugar una serie completa de instrucciones SQL (seguidas por un COMMIT o un ROLLBACK explícito) como una sola transacción. Sin embargo, esta práctica no es muy recomendable, ya que puede ocasionar que partes de la base de datos permanezcan bloqueadas, y por lo tanto inaccesibles, para otros usuarios durante periodos excesivos (vea el capítulo 15). Además, en tales ambientes es posible que los usuarios finales queden *bloqueados irreversiblemente* entre sí, lo cual es otro buen argumento para no autorizar esta práctica (nuevamente vea el capítulo 15).

EJERCICIOS

14.1 Los sistemas no permiten que una determinada transacción confirme los cambios a las bases de datos (ni a las varrels ni...) en forma individual; es decir, sin confirmar simultáneamente los cambios a todas las demás bases de datos (o varrels o...). ¿Por qué no?

14.2 Las transacciones no pueden estar anidadas unas dentro de otras. ¿Por qué no?

14.3 Explique la regla de la escritura anticipada de la bitácora. ¿Por qué esta regla es necesaria?

14.4 ¿Qué consecuencias tiene en la recuperación si:

a. Fuerza los búferes hacia la base de datos en el momento del COMMIT?

b. Nunca escribe físicamente los búferes en la base de datos antes del COMMIT?

14.5 Explique el protocolo de confirmación de dos fases, así como las implicaciones de una falla en la parte (a) del coordinador, (b) de un participante durante cada una de las dos fases.

14.6 Usando la base de datos de proveedores y partes, escriba un programa SQL para leer e imprimir todas las partes en orden según su número, borrando la décima parte conforme avanza y comenzando una nueva transacción después de cada décima fila. Puede dar por hecho que la regla DELETE de clave externa para las partes de los envíos, especifica CASCADE (es decir, para efectos de este ejer cicio usted puede ignorar los envíos). *Nota:* Pedimos específicamente una solución SQL tal que pueda usar el mecanismo de cursor SQL en la respuesta.

REFERENCIAS Y BIBLIOGRAFÍA

14.1 Philip A. Bernstein: "Transaction Processing Monitors", *CACM 33,* No. 11 (noviembre, 1990).

Cita: "Un *sistema PT* es un conjunto integrado de productos que... incluyen tanto hardware (procesadores, memorias, discos y controladores de comunicaciones) como software (sistemas operativos, sistemas de administración de bases de datos, redes de computadoras y monitores PT). Gran parte de la integración de estos productos es proporcionada por los monitores PT." El artículo sirve como una introducción informal hacia la estructura y funcionalidad de los monitores PT.

14.2 Philip A. Bernstein, Vassos Hadzilacos y Nathan Goodman: *Concurrency Control and Recovery in Database Systems.* Reading, Mass.: Addison-Wesley (1987).

Como lo indica el título, es un libro de texto que trata no sólo la recuperación, sino la administración completa de transacciones, desde una perspectiva mucho más formal que el presente capítulo.

14.3 A. Bilris *et al.:* "ASSET: A System for Supporting Extended Transactions", Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, Minn, (mayo, 1994).

Como están descritas en el cuerpo de este capítulo y el siguiente, las nociones básicas de transacciones están consideradas como demasiado rígidas para determinados tipos de aplicaciones recientes (en especial las aplicaciones altamente interactivas) y por lo tanto, se ha propuesto una variedad de "modelos de transacciones extendidas" para abordar este asunto (vea la referencia [14.15]). Sin embargo, hasta estos momentos, ninguna de estas propuestas ha mostrado ser realmente superior a las demás y, por consecuencia, "los fabricantes de bases de datos [se han resistido a] incorporar cualquiera de estos modelos en un producto".

El enfoque de ASSET es algo diferente. En vez de proponer un nuevo modelo de transacciones, ofrece un conjunto de operadores primitivos (entre los que se encuentra el COMMIT usual y los demás), así como otros nuevos que pueden ser usados para "definir modelos de transacción personalizados adecuados para aplicaciones específicas". En particular, el artículo muestra la manera en que se puede usar ASSET para especificar "transacciones anidadas, transacciones divididas, sagas y otros modelos de transacciones extendidas que se describen en la literatura".

14.4 L. A. Bjork: "Recovery Scenario for DB/DC System", Proc. ACM National Conf., Atlanta, Ga. (agosto, 1973).

Este artículo y su complemento, escrito por Davies [14.7], representan probablemente el primer trabajo teórico en el área de recuperación.

14.5 R. A. Crus: "Data Recovery in IBM DATABASE 2", *IBM Sys. J. 23,* No. 2 (1984).

Describe con detalle el mecanismo de recuperación de DB2, y al hacerlo proporciona una buena descripción de las técnicas de recuperación en general. En particular, el artículo explica la mane ra en que DB2 se recupera de una caída del sistema durante el propio proceso de recuperación, mientras algunas transacciones están ejecutando una instrucción deshacer. Este problema requie re un cuidado especial para asegurar que las actualizaciones no confirmadas de la transacción que se está deshaciendo, efectivamente se deshagan (en cierto sentido, lo opuesto al problema de la actualización perdida, vea el capítulo 15).

14.6 C. J. Date: "Distributed Database: A Closer Look", en C. J. Date y Hugh Darwen, *Relational Da tabase Writings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).

La sección 14.6 del presente capítulo describe lo que podría llamarse el protocolo *básico* de confirmación de dos fases. Es posible hacer varias mejoras al protocolo base. Por ejemplo, si en la fase 1 el participante *P* responde al coordinador C que no hizo actualizaciones en la transacción que está en consideración (es decir, fue de *sólo lectura),* entonces *C* puede simplemente ignorar a *P* en la fase 2. Además, si en la fase 1 todos los participantes responden que son de *sólo lectura* es posible omitir completamente la fase 2.

Es posible implementar otras mejoras y refinamientos. Este artículo incluye un tutorial que describe algunas de ellas. Específicamente trata los protocolos de *confirmación presupuesta y de instrucción deshacer presupuesta* (versiones mejoradas del protocolo básico), el modelo de *árbol de procesos* (cuando un participante necesita servir como coordinador para determinadas partes de una transacción) y lo que pasa cuando ocurre una/aWa *de comunicación* durante el proceso de reconocimiento entre un participante y el coordinador. *Nota:* De hecho, aunque las explicaciones son presentadas en el contexto de un sistema distribuido, la mayoría de los conceptos tienen un campo de aplicación más amplio. Consulte el capítulo 20 para una explicación más amplia de algunos de estos temas.

14.7 C. T. Davies, Jr.: "Recovery Semantics for a DB/DC System", Proc. ACM National Conf., Atlanta, Ga. (agosto, 1973).

Vea el comentario de la referencia [14.4].

14.8 C. T. Davies, Jr.: "Data Processing Spheres of Control", *IBM Sys. J. 17,* No. 2 (1978).

Las *esferas de control* fueron el primer intento para investigar y formalizar lo que posteriormente llegó a ser la disciplina de la administración de transacciones. Una esfera de control es una abstracción que representa una parte del trabajo que (desde afuera) puede ser vista como atómica. Sin embargo, a diferencia de las transacciones como las que son soportadas en la mayoría de los sistemas actuales, las esferas de control pueden estar anidadas una dentro de otra a una profundidad arbitraria (vea la respuesta al ejercicio 14.2).

14.9 Héctor García-Molina y Kenneth Salem: "Sagas", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

El mayor problema con las transacciones, tal como se describen en el cuerpo de este capítulo, es que tácitamente tienen una duración muy corta (milisegundos o incluso microsegundos). Si una transacción dura mucho tiempo (horas, días o semanas), entonces: (a) si tiene que ser deshecha es necesario deshacer una gran cantidad de trabajo y (b) aunque sea satisfactoria aún tiene que conservar recursos del sistema (datos de la base de datos, etcétera) por un tiempo excesivamente largo, lo que bloqueará a los demás usuarios (vea el capítulo 15). Desgraciadamente, muchas transacciones del "mundo real" tienden a durar mucho, en especial en algunas de las áreas de aplicación más recientes, tales como la ingeniería de hardware y de software.

Las *sagas* atacan este problema. Una saga es una secuencia de transacciones cortas (en el sentido usual del término) con la propiedad de que el sistema garantiza *que* (a) todas las transacciones de la secuencia se ejecutan satisfactoriamente *o* (b) determinadas transacciones compensatorias son ejecutadas para cancelar los efectos de las transacciones terminadas satisfactoriamente en una ejecución incompleta de toda la saga (por lo tanto, haciendo como si la saga nunca se hubiera ejecutado). En un sistema bancario, por ejemplo, podemos tener la transacción "sumar $100 a la cuenta A", y la transacción compensatoria sería obviamente "restar $100 de la cuenta A". Una extensión a la instrucción COMMIT permite que el usuario informe al sistema cuál es la transacción compensatoria a ejecutar en caso de que posteriormente sea necesario cancelar los efectos de la transacción que ahora se termina. Observe que, idealmente, ¡una transacción compensatoria nunca debe terminar con una instrucción deshacer!.

14.10 James Gray: "Notes on Data Base Operating Systems", en R. Bayer, R. M. Graham y G. Seegmuller (eds.), *Operating Systems: An Advanced Course* (Springer Verlag *Lecture Notes in Computer Science 60).* Nueva York, N.Y.: Springer Verlag (1978). También disponible como IBM Research Report RJ 2188 (febrero, 1978).

Es una de las primeras fuentes y con seguridad una de las más accesibles sobre la administración de transacciones. Contiene la primera descripción general del protocolo de confirmación de dos fases. Obviamente no está tan completa como la referencia 14.12 que es más reciente, sin embargo aún es recomendable.

14.11 Jim Gray: "The Transaction Concept: Virtues and Limitations", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, Francia (septiembre, 1981).

Es una declaración concisa de diversos conceptos y problemas relacionados con las transacciones e incluye una variedad de aspectos de la implementación. Un problema particular que trata es el siguiente: las transacciones, como generalmente se entienden, no pueden estar anidadas dentro de otras (vea la respuesta al ejercicio 14.2). Sin embargo, ¿no habrá alguna forma para permitir que las transacciones se dividan en "subtransacciones" más pequeñas? La respuesta es un "sí" con limitaciones, ya que es posible que una transacción establezca *puntos de resguardo* intermedios mientras está en ejecución y posteriormente ejecute la instrucción deshacer hasta un punto de resguardo establecido anteriormente (en caso de que lo requiera), en vez de tener que ejecutar la instrucción deshacer hasta el inicio. De hecho, se ha incorporado esta facilidad en varios sistemas implementados, incluyendo por ejemplo a Ingres (el producto comercial y no el prototipo) y a System R (aunque no DB2). En general, este concepto se acerca mucho a la noción de transac-

ciones en la forma en que se comprende este término en el mundo real. Pero observe que el establecimiento de un punto de resguardo no es lo mismo que la realización de un COMMIT, ya que las actualizaciones realizadas por la transacción permanecen invisibles ante las demás transacciones, hasta el final de la transacción (satisfactoria).

Nota: Las "sagas" de la referencia [14.9], que en algunos aspectos tratan el mismo problema que los puntos de resguardo, fueron propuestas después de que Gray escribiera este artículo por primera vez.

14.12 Jim Gray y Andreas Reuter: *Transaction Processing: Concepts and Techniques.* San Mateo, Calif.: Morgan Kaufmann (1993).

Si alguna vez algún texto de las ciencias de la computación mereció el epíteto de "clásico instantáneo", es éste. Al principio su tamaño puede parecer amenazante (?), pero los autores muestran una ligereza envidiable que hace que hasta los aspecto más áridos del tema sean de lectura agradable. En el prefacio establecen su propósito de "ayudar... a resolver problemas reales". El libro es "pragmático, al tratar los puntos básicos de las transacciones con gran detalle", y la presentación "está llena de fragmentos de código que muestran... algoritmos básicos y estructuras de datos" sin ser "enciclopédico". A pesar de esta última advertencia, el libro es bastante completo (sin sorprender) y seguramente está destinado a convertirse en el estándar. Lo recomiendo ampliamente.

14.13 Jim Gray *et al.:* "The Recovery Manager of the System R Data Manager", *ACM Comp. Surv. 13,* No. 2 (junio, 1981).

Las referencias [14.13] y [14.18] tratan las características de recuperación del System R (que en cierta forma fue pionero en este campo). La referencia [14.13] proporciona un panorama general del subsistema de recuperación completo, y la referencia [14.18] describe con detalle un aspecto específico, llamado el mecanismo de *página de sombra* (vea más adelante el comentario a esta última referencia).

14.14 Theo Harder y Andreas Reuter: "Principles of Transaction-Oriented Database Recovery", *ACM Comp. Surv. 15,* No. 4 (diciembre, 1983).

Al ser la fuente del acrónimo ACID, este artículo presenta un tutorial muy claro y cuidadoso sobre los principios de la recuperación. También proporciona un marco de referencia terminológico consistente para describir una amplia variedad de esquemas de recuperación y técnicas de registro uniforme en bitácoras; también clasifica y describe varios de los sistemas existentes de acuerdo con este marco de referencia.

El artículo incluye algunas cifras empíricas interesantes con relación a la frecuencia de ocurrencia y a los tiempos de recuperación típicos (aceptables) para los tres tipos de fallas (local, de sistema y de medio) en un sistema grande típico:

Tipo de falla	*Frecuencia de ocurrencia*	*Tiempo de recuperación*
Local	10 a 100 por minuto	El mismo que el tiempo de ejecución de la transacción
De sistema	Varias por semana Una	
Del medio	o dos por año	Unos cuantos minutos De una a dos horas

14.15 Henry F. Korth: "The Double Life of the Abstraction Concept: Fundamental Principle and Evolving System Concept" (invited talk), Proc. 21st Int. Conf. on Very Large Data Bases, Zurich, Suiza (septiembre, 1995).

Presenta un panorama general conciso sobre la forma en que el concepto de transacción necesita evolucionar para soportar nuevos requerimientos de aplicaciones.

14.16 Henry F. Korth, Eliezer Levy y Abraham Silberschatz: "A Formal Approach to Recovery by Compensating Transactions", Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (agosto, 1990).

Formaliza la noción de **transacción compensatoria** usada en las sagas [14.9] y en muchos otros casos, para "deshacer" transacciones confirmadas (así como las no confirmadas).

14.17 David Lomet y Mark R. Tuttle: "Redo Recovery after System Crashes", Proc. 21st Int. Conf. on Very Large Data Bases, Zurich, Suiza (septiembre, 1995).

Es un análisis preciso y cuidadoso de la recuperación mediante rehacer (es decir, la recuperación hacia delante). "[Aunque] la recuperación mediante rehacer es simplemente una forma de recuperación, es... importante [debido a que es una parte crucial del proceso general de recuperación y] debe resolver los problemas más difíciles". (En este sentido observe que, a diferencia del algoritmo que se muestra en la sección 14.4, ARIES [14.19] "sugiere la comprensión de la recuperación... como la realización de una recuperación mediante rehacer seguida de una recuperación mediante deshacer".) Los autores mencionan que su análisis conduce hacia una mejor comprensión de las implementaciones existentes y tiene el potencial para sistemas de recuperación significativamente mejorados.

14.18 Raymond A. Lorie: "Physical Integrity in a Large Segmented Database", *ACM TODS* 2, No. 1 (marzo, 1977).

Como expliqué en el comentario de la referencia [14.13], este artículo trata un aspecto específico del subsistema de recuperación del System R, llamado mecanismo de *página de sombra*. (Observe de paso que el término "integridad", como es usado en el título de este artículo, tiene muy poco que ver con la noción de integridad que traté en el capítulo 8.) La idea básica es simple: cuando una actualización (no confirmada) se escribe por primera vez en la base de datos, el sistema no sobrescribe la página existente sino que guarda una nueva página en cualquier otro lugar del disco. La página antigua es entonces la "sombra" de la nueva. La confirmación de la actualización involucra actualizar diversos apuntadores para que apunten hacia la nueva página y descarten la de sombra; por otro lado, deshacer la actualización involucra reintegrar la página de sombra y deshacer la nueva.

Aunque conceptualmente es simple, el esquema de página de sombra tiene la seria desventaja de que destruye cualquier agrupamiento físico que pudiera haber existido previamente en los datos. Por esta razón el esquema no se tomó del System R para usarlo en DB2 [14.5], aunque *fue* usado en SQL/DS [4.13].

14.19 C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh y Peter Schwartz: "ARIES: A Tran saction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write Ahead Logging", *ACM TODS 17,* No. 1 (marzo, 1992).

ARIES significa "algoritmo para la recuperación y el aislamiento explotando la semántica". ARIES ha sido implementado ("en diversos grados") en muchos sistemas comerciales y experimentales, incluyendo al DB2 en particular. Citando al artículo: "Las soluciones [para el problema de la administración de transacciones] pueden juzgarse usando varias medidas: grado de concurrencia soportado dentro de una página y a través de páginas, complejidad de la lógica resultante, sobrecarga de espacio en almacenamiento no volátil y en memoria para los datos y la bitácora, sobrecarga en términos de la cantidad de E/S sincrónica y asincrónica requerida durante el reinicio/recuperación y procesamiento normal, tipos de funcionalidad soportada (deshacer transacciones parciales, etcétera), cantidad de procesamiento realizado durante el reinicio/recuperación, grado de procesamiento concurrente soportado durante el reinicio/recuperación, alcance de las transacciones deshechas inducidas por el sistema a causa de bloqueos mortales, restricciones impuestas sobre los datos almacenados (por ejemplo, el requerimiento de claves únicas para todos los registros, la restricción del tamaño máximo de los objetos al tamaño de la página, etcétera), capacidad para soportar nuevos tipos de bloqueo que permitan la ejecución concurrente de operaciones como el incremento o decremento del mismo dato por transacciones diferentes —con base en la conmutatividad y otras propiedades—, etcétera. [A ARIES] le va muy bien con respecto a estas medidas". (Cambiando un poco las palabras).

Desde que ARIES fue diseñado, se han desarrollado y descrito numerosos ajustes y versiones especializadas en la literatura: ARIES/CS (para sistemas cliente-servidor), ARIES/IM (para administración de índices), ARIES/NT (para transacciones anidadas), etcétera.

UESTAS A EJERCICIOS SELECCIONADOS

14.1 Esta característica entraría en conflicto con el objetivo de la atomicidad de transacciones. Si una transacción pudiera confirmar solamente algunas, aunque no todas sus actualizaciones, entonces las que no fueran confirmadas podrían ser deshechas subsecuentemente y, en cambio, las confirmadas no podrían serlo. Entonces la transacción ya no sería "todo o nada".

14.2 Esta característica entraría en conflicto con el objetivo de la atomicidad de las transacciones. Considere lo que podría pasar si la transacción *B* estuviera anidada dentro de la transacción *A* y ocurriera la siguiente secuencia de eventos (nuevamente suponga, por simplicidad, que tiene sentido hablar acerca de "actualización de una tupia"):

```
BEGIN TRANSACTION (transacción A) ;

    BEGIN TRANSACTION (transacción S) ;
    la transacción S actualiza la tupia t ;
    COMMIT (transacción S) ;

ROLLBACK (transacción A) ;
```

Si en este momento restauramos la tupia *t* al valor que tenía antes de *A*, entonces el COMMIT de *B* no sería en realidad un COMMIT. En forma inversa, si el COMMIT de *B* fuera genuino, la tupia *t* no podría ser restaurada a su valor anterior a A y por lo tanto, no podría realizarse el ROLLBACK de *A*.

Observe que decir que las transacciones no se pueden anidar significa que un programa puede ejecutar una operación BEGIN TRANSACTION sólo cuando no tiene ninguna transacción en ejecución.

De hecho, muchos autores (comenzando por Davies en la referencia [14.7]) han propuesto la posibilidad de anidar transacciones eliminando el requerimiento de *durabilidad* (la propiedad "D" de ACID) en la parte de la transacción interna. Esto significa que el COMMIT de una transacción interna confirmará las actualizaciones de la transacción *pero sólo hasta el siguiente nivel externo.* Si posteriormente ese nivel externo termina con deshacer, la transacción interna también será deshecha. En el ejemplo, el COMMIT de *B* será entonces un COMMIT solamente para *A*, no para el exterior, y podrá ser revocado subsecuentemente.

Vale la pena hacer una pausa para pensar en las ideas anteriores. Las transacciones anidadas pueden pensarse como una generalización de los puntos de resguardo [14.11]. Los puntos de resguardo permiten que una transacción sea organizada como una *secuencia* de acciones (y es posible deshacer las transacciones en cualquier momento hasta el inicio de cualquier acción anterior en la secuencia). Por el contrario, el anidamiento permite que una transacción se organice, en forma recursiva, como una *jerarquía* de estas acciones (vea la figura 14.4). En otras palabras:

- BEGIN TRANSACTION se extiende para soportar subtransacciones (es decir, si BEGIN TRANS ACTION se emite cuando una transacción ya está ejecutándose, inicia a una transacción *hija);*
- COMMIT "confirma", pero sólo dentro del *alcance de la madre* (en caso de que la transacción sea una hija);
- ROLLBACK deshace el trabajo, pero sólo regresa hasta el inicio de esta transacción particular (incluyendo transacciones hijas, nietas, etcétera, pero sin incluir la transacción madre, en caso de existir).

Hacemos notar que las transacciones anidadas son difíciles de implementar, desde un punto de vista puramente sintáctico, en un lenguaje como SQL al que le falta el BEGIN TRANSACTION explícito

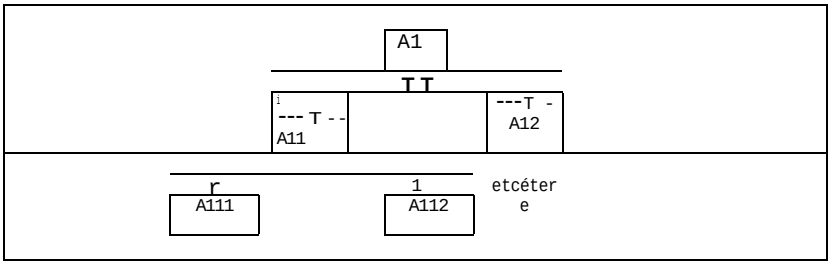


Figura 14.4 Una transacción anidada típica.

(tiene que haber *alguna* forma explícita para indicar el inicio de una transacción interna y marcar el punto a deshacer en caso de que falle la transacción interna).

14.4

a. Nunca es necesario rehacer después de una falla de sistema.

b. La acción física de deshacer nunca es necesaria y por lo tanto, tampoco son necesarios los registros de bitácora para deshacer.

14.6 Este ejercicio es típico de una gran clase de aplicaciones, y la siguiente solución también es típica.

```
EXEC SQL DECLARE CP CURSOR FOR
         SELECT P.P#, P.PARTE, P.COLOR, P.PESO, P.CIUDAD
         FROM P
WHERE P.P# > P#_anterior ORDER BY P#
; P#_anterior := '     ' ¡ eof :«
falso ; DO WHILE ( eof = falso ) ;
EXEC SQL OPEN CP ¡
   DO cuenta := 1 TO 10 ;
      EXEC SQL FETCH CP INTO :P#, ... ;
      IF SQLSTATE ■ '02000' THEN DO ;
         EXEC SQL CLOSE CP ;
         EXEC SQL COMMIT ; eof :
         = verdadero ; END DO ;
   ELSE imprimir P#, ... ; END
   IF ; END DO ;
EXEC SQL DELETE FROM P WHERE P.P# = :P# ;
EXEC SQL CLOSE CP ; EXEC SQL COMMIT ;
P#_anterior := P# ; END DO ;
```

Observe que al final de cada transacción perdemos la posición dentro de la tabla de partes P (aunque no se cierre explícitamente el cursor CP, el COMMIT lo cerrará automáticamente). Por lo tanto, el código anterior no será muy eficiente, ya que cada nueva transacción requiere una búsqueda en la tabla de partes para volver a establecer la posición. Podríamos mejorar un poco las cosas si hubiera un índice sobre la columna P# —como es probable que ocurra puesto que {P#} es la clave primaria— y el optimizador escogiera ese índice como la ruta de acceso para la tabla.

CAPITULO 15

:urrencia

PRODUCCIÓN

Como expliqué en la introducción del capítulo 14, el cual trata sobre la recuperación, los temas de la concurrencia y la recuperación van de la mano y ambos son parte del tema más general de la *administración de transacciones.* Ahora pondremos nuestra atención en la concurrencia. Por lo general, el término **concurrencia** se refiere al hecho de que los DBMSs (Sistemas de Administración de Bases de Datos) permiten que muchas transacciones accedan a una misma base de datos a la vez. Como bien es sabido, en un sistema de éstos se necesita algún tipo de **mecanismo de control de concurrencia** para asegurar que las transacciones concurrentes no interfieran entre sí. Posteriormente, en la sección 15.2, muestro ejemplos de los tipos de interferencias que pueden ocurrir (en ausencia de un control adecuado). La estructura del capítulo es la siguiente:

- Como acabo de decir, la sección 15.2 explica algunos de los problemas que pueden pre sentarse si no se proporciona el control de concurrencia adecuado.
- La sección 15.3 presenta el mecanismo convencional para el manejo de dichos problemas, al cual se llama **bloqueo.** *Nota:* El bloqueo no es el único enfoque posible para el problema del control de la concurrencia, pero es por mucho el más comúnmente encontrado en la práctica. Describo algunos otros enfoques en los comentarios de las referencias [15.1], [15.3], [15.6], [15.7], [15.14] y [15.15].
- La sección 15.4 muestra la manera en que se puede usar el bloqueo para resolver los pro blemas descritos en la sección 15.2.
- Por desgracia, el bloqueo presenta sus propios problemas; de ellos, el más conocido es el **bloqueo mortal** (también conocido como abrazo mortal). La sección 15.5 trata este tema.
- La sección 15.6 describe el concepto de **seriabilidad,** que es reconocido generalmente como el criterio formal de corrección para la ejecución de un conjunto de transacciones concurrentes.
- Las secciones 15.7 y 15.8 consideran algunas precisiones importantes sobre la idea básica del bloqueo, a las que se llama **niveles de aislamiento y bloqueo por aproximación.**
- La sección 15.9 describe las características relevantes de SQL.
- Por último, la sección 15.10 presenta un resumen y algunas conclusiones.

Nota: Aquí aplico nuevamente algunas anotaciones generales de la introducción del capítulo 14:

Primero, las ideas de concurrencia, al igual que las de recuperación, son más bien independientes de si el sistema subyacente es relacional o de algún otro tipo. Sin embargo, es importante hacer notar que, al igual que en la recuperación, casi todo el trabajo teórico inicial en el área fue realizado en un contexto específicamente relacional "por seguridad" [15.5].

Segundo, la concurrencia, al igual que la recuperación, es un tema muy amplio y todo lo que podemos esperar en este capítulo es la presentación de algunas de las ideas más importantes y básicas. Los ejercicios, respuestas y comentarios sobre referencias que se encuentran al final del capítulo, incluyen algunas explicaciones sobre determinados aspectos más avanzados del tema.

15.2 TRES PROBLEMAS DE CONCURRENCIA

Comencemos considerando alguno de los problemas que debe resolver cualquier mecanismo de control de concurrencia. Existen esencialmente tres maneras en las que las cosas pueden salir mal; esto es, tres formas en las que una transacción, aunque sea correcta por sí misma, puede producir una respuesta incorrecta si alguna otra transacción interfiere con ella en alguna forma. Observe que la transacción que interfiere también puede ser correcta por sí misma; lo que produce el resultado incorrecto general es el *intercalado* sin control entre las operaciones de las dos transacciones correctas. (Por lo que se refiere al hecho de que una transacción sea "correcta por sí misma", lo único que queremos decir es que la transacción no viola la *regla de oro,* vea el capítulo 8.) Los tres problemas son:

- El problema de la *actualización perdida,*
- El problema de la dependencia no confirmada y
- El problema del análisis inconsistente.

Los consideraremos de uno en uno.

El problema de la actualización perdida

Considere la situación que se ilustra en la figura 15.1. Esta figura debería leerse de la siguiente forma: la transacción *A* recupera alguna tupia ***t*** en el tiempo *ti;* la transacción *B* recupera la

Transacción A	*Tiempo*	*Transacción B*
RECUPERAR *t*	***ti***	
	f	RECUPERAR t
ACTUALIZAR t	**f**	
		ACTUALIZAR í

Figura 15.1 La transacción *A* pierde una actualización en el tiempo í4.

misma tupia *t* en el tiempo *t2;* la transacción A actualiza la tupia en el tiempo *t3* (con base en los valores vistos en el tiempo *ti),* y la transacción *B* actualiza la misma tupia en el tiempo *t4* (con base en los valores vistos en el tiempo í2, que son los mismos vistos en el tiempo *ti).* La actualización de la transacción A se pierde en el tiempo *t4,* ya que la transacción *B* la sobrescribe sin siquiera mirarla. *Nota:* Aquí, y a lo largo del capítulo, adoptamos nuevamente la idea de que tiene sentido hablar sobre "actualizar una tupia".

El problema de la dependencia no confirmada

El problema de la dependencia no confirmada se presenta al permitir que una transacción recupere, o lo que es peor, actualice una tupia que ha sido actualizada por otra transacción pero que aún no ha sido confirmada por esa misma transacción. Puesto que no ha sido confirmada, sigue existiendo la posibilidad de que nunca lo sea y de que en su lugar se deshaga mediante un ROLLBACK; en cuyo caso, la primera transacción habrá visto datos que ya no existen (y en cierto sentido nunca existieron). Considere las figuras 15.2 y 15.3.

En el primer ejemplo (figura 15.2) la transacción A ve una *actualización no confirmada* en el tiempo *t2* (llamada también *cambio no confirmado).* Esta actualización es posteriormente deshecha en el tiempo *t3.* Por lo tanto, la transacción A está operando sobre una suposición falsa;

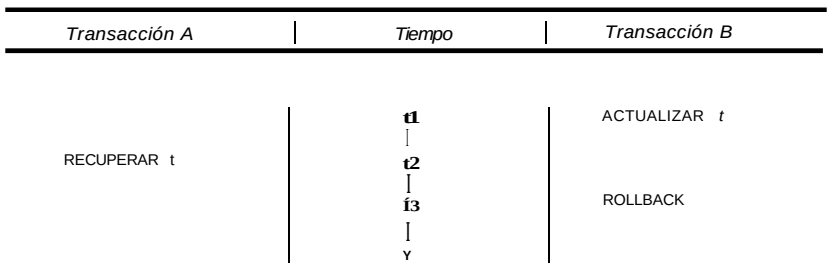


Figura 15.2 La transacción *A* llega a ser dependiente de un cambio no confirmado en el tiempo f2.

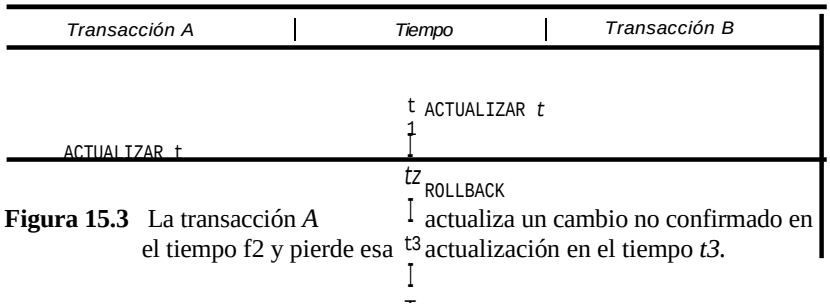


Figura 15.3 La transacción *A* actualiza un cambio no confirmado en el tiempo f2 y pierde esa actualización en el tiempo *t3.*

es decir, la suposición de que la tupia *t* tiene el valor visto en el tiempo *t2,* siendo que tiene el valor que tenía antes del tiempo *ti.* Por consecuencia, la transacción *A* producirá una salida incorrecta. Observe además que el hecho de que la transacción *B* se haya deshecho probablemente no se deba a una falla de *B,* sino que (por ejemplo) puede ser el resultado de una caída del sistema. (Y podría ser que la transacción *A* ya hubiera terminado en ese momento; en cuyo caso, la falla no permitiría una instrucción ROLLBACK para *A.)*

El segundo ejemplo (figura 15.3) es aún peor. La transacción *A* no sólo llega a ser dependiente del cambio no confirmado en el tiempo *t2,* sino que de hecho pierde una actualización en el tiempo *t3,* debido a que la instrucción ROLLBACK en ese tiempo hace que la tupia *t* sea restaurada a su valor anterior al tiempo *ti.* Esta es otra versión del problema de la actualización perdida.

El problema del análisis inconsistente

Considere la figura 15.4, que muestra dos transacciones, *A y B,* operando sobre tupias de una cuenta (ACC): la transacción *A* está sumando saldos de cuenta y la transacción *B* está transfiriendo una cantidad de 10 de la cuenta 3 a la cuenta 1. El resultado de 110 producido por *A* es obviamente incorrecto; y si *A* continuara y escribiera ese resultado en la base de datos, dejaría

ACC 1	ACC 2	ACC 3
40	50	30

Transacción A	*Tiempo*	*Transacción B*
RECUPERAR ACC 1 suma = 40	t1	
RECUPERAR ACC 2 suma ■ 90		
	í3	RECUPERAR ACC 3
	í4	ACTUALIZAR ACC 3 30 —►■ 20
	í5	RECUPERAR ACC 1
	í6	ACTUALIZAR ACC 1 40 -- »• 50
	í7	COMMIT
RECUPERAR ACC 3 : suma = 110, y *no* 120	t8	

Figura 15.4 La transacción *A* realiza un análisis inconsistente.

en efecto a la base de datos en un estado inconsistente.* Decimos que *A* ha visto un estado inconsistente de la base de datos y por lo tanto, ha realizado un análisis inconsistente. Observe la diferencia entre este ejemplo y el anterior: aquí no hay posibilidad de que *A* sea dependiente de un cambio no confirmado, ya que *B* confirma todas sus actualizaciones antes de que *A* vea a ACC3.

15.3 BLOQUEO

Como dije en la sección 15.1, todos los problemas de la sección 15.2 pueden ser resueltos por medio de una técnica de control de concurrencia llamada **bloqueo.** La idea básica es simple: cuando una transacción deba asegurarse de que algún objeto en el que está interesada —por lo general, una tupia de base de datos— no cambiará de ninguna forma mientras lo esté usando, **adquiere un bloqueo** sobre ese objeto. El efecto del bloqueo es "inhibir todas las demás transacciones" en ese objeto y por lo tanto, impedir que lo cambien. Por lo tanto, la primera transacción es capaz de realizar todo su procesamiento con el conocimiento certero de que el objeto en cuestión permanecerá en un estado estable durante todo el tiempo que ésta lo desee.

Ahora daremos una explicación más detallada sobre la forma en que funcionan los bloqueos:

1. Primero suponemos que el sistema soporta dos tipos de bloqueos, bloqueos **exclusivos** (X) y bloqueos **compartidos** (S), los cuales defino en los siguientes párrafos. *Nota:* En oca siones, a los bloqueos X y S se les llama bloqueos **de escritura y de lectura,** respectiva mente. Suponemos, mientras no se especifique otra cosa, que los bloqueos X y S son los únicos disponibles; vea la sección 15.8 para ejemplos de otros tipos. También suponemos que las tupias son el único tipo de "objeto bloqueable"; nuevamente, vea la sección 15.8 para otras posibilidades.
2. Si la transacción *A* pone un bloqueo exclusivo (X) sobre la tupia *t,* entonces se rechazará una petición de cualquier otra transacción *B* para un bloqueo de cualquier tipo sobre la tupia *t.*
3. Si la transacción *A* pone un bloqueo compartido (S) sobre la tupia *t* entonces:
   - Se rechazará una petición de cualquier otra transacción *B* para un bloqueo X sobre *t.*
   - Se otorgará una petición de cualquier otra transacción *B* para un bloqueo S sobre *t* (esto es, ahora también *B* tendrá un bloqueo S sobre *i).*

Estas reglas pueden ser resumidas adecuadamente por medio de una *matriz de compatibilidad de tipos de bloqueo* (figura 15.5). Esta matriz es interpretada de la siguiente forma: considere alguna tupia *t;* suponga que la transacción *A* tiene actualmente un bloqueo sobre *t,* como lo indican las entradas de los encabezados de columna (un guión indica que no hay bloqueo), y suponga que otra transacción *B* emite una petición de bloqueo sobre *t* como lo indican las entradas del lado izquierdo (para complementar la explicación, incluimos el caso en que no hay bloqueo). Un "No" significa un *conflicto* (la petición de *B* no puede ser satisfecha y *B* pasa a un **estado de espera)** y un "Sf' indica compatibilidad (la petición de *B* es satisfecha). Obviamente la matriz es simétrica.

---

* Con relación a esta posibilidad (es decir, la escritura del resultado en la base de datos), es necesario suponer que no hay alguna restricción de integridad que impida tal escritura.

	X	S	-
X	No	No	Sí
S	No	Sí	Sí
-	Sí	Sí	Sí

Figura 15.5 Matriz de compatibilidad para los tipos de bloqueo X y S.

A continuación presentamos un **protocolo de acceso a datos (o protocolo de bloqueo)** que utiliza los bloqueos X y S, tal como fueron definidos, para garantizar que no ocurran los problemas que describo en la sección 15.2:*

1. Una transacción que desea recuperar una tupia debe primero adquirir un bloqueo S sobre esa tupia.
2. Una transacción que desea actualizar una tupia primero debe adquirir un bloqueo X sobre esa tupia. En forma alterna, si ya tiene un bloqueo S sobre la tupia (como ocurre en una se cuencia de RECUPERACIÓN-ACTUALIZACIÓN) entonces debe *promover* ese bloqueo S hacia el nivel X.

Nota: Las peticiones de las transacciones para bloqueos sobre tupias normalmente están implícitas; ya que una petición de "recuperación de tupia" es una petición implícita de un bloqueo S, y una petición de "actualización de tupia" es una petición implícita de un bloqueo X sobre la tupia relevante. Por supuesto (como siempre), también tomamos el término "actualizar" para incluir los INSERTS y DELETEs, así como las *propias* actualizaciones, pero las reglas requieren algunos ajustes menores para encargarse de los INSERTs y DELETEs. Aquí omitimos los detalles.

Para continuar con el protocolo:

3. Si una petición de bloqueo de una transacción *B* es rechazada porque entra en conflicto con un bloqueo que ya tiene la transacción *A,* la transacción *B* pasa a un estado de espera y per manecerá así hasta que se libere el bloqueo de *A. Nota:* El sistema debe garantizar que *B* no quede en espera por siempre (una posibilidad a la que a veces se le llama espera **in definida).** Una forma simple para proporcionar esta garantía es dar servicio a todas las peti ciones de bloqueo con base en "la primera que llega es la primera atendida".
4. Los bloqueos X se mantienen hasta el final de la transacción (COMMIT o ROLLBACK). Los bloqueos S también se mantienen normalmente hasta ese momento (pero vea la sección 15.7).

15.4 OTRA VEZ LOS TRES PROBLEMAS DE CONCURRENCIA

Ahora podemos ver la forma en que el esquema anterior resuelve los tres problemas descritos en la sección 15.2. Nuevamente los consideraremos de uno en uno.

---

* El protocolo descrito es un ejemplo del *bloqueo de dos fases* (y se trata con detalle en la sección 15.6).

El problema de la actualización perdida

La figura 15.6 es una versión modificada de la figura 15.1, que muestra lo que le pasaría a la ejecución intercalada de esa figura bajo el protocolo de bloqueo descrito en la sección 15.3. La actualización de la transacción *A* en el tiempo *t3* no es aceptada, ya que es una petición implícita de un bloqueo X sobre *t,* y esta petición entra en conflicto con el bloqueo S que ya tiene la transacción *B* y, por lo tanto, *A* pasa a un estado de espera. Por razones similares, *B* entra en un estado de espera en el tiempo *t4.* Ahora ambas transacciones no pueden continuar, por lo que ya no existe la posibilidad de perder alguna actualización. El bloqueo, por lo tanto, resuelve el problema de la actualización perdida ¡convirtiéndolo en otro problema!, pero al menos resuelve el problema original. Al nuevo problema se le llama **bloqueo mortal.** Este problema es tratado en la sección 15.5.

Transacción A	*Tiempo*	*Transacción B*
RECUPERAR t	*t1*	—
(adquiere bloqueo S sobre t)		
-	*t2*	RECUPERAR t
—		(adquiere bloqueo S sobre *t)*
ACTUALIZAR t	*t3*	-
(solicita bloqueo X sobre t)		-
espera		—
espera	M	ACTUALIZAR t
espera		(solicita bloqueo X sobre *t)*
espera		espera
espera		espera
espera		espera

Figura 15.6 No se pierde ninguna actualización, pero ocurre un bloqueo mortal en el tiempo *t4.*

El problema de la dependencia no confirmada

Las figuras 15.7 y 15.8 son versiones modificadas de las figuras 15.2 y 15.3 respectivamente, las cuales muestran lo que sucedería con las ejecuciones intercaladas de aquellas figuras bajo el protocolo de bloqueo de la sección 15.3. La operación de la transacción *A* en el tiempo í2 (RECUPERAR en la figura 15.7 y ACTUALIZAR en la figura 15.8) no se acepta en ningún caso, ya que es una petición implícita de un bloqueo sobre *t,* y esta petición entra en conflicto con el bloqueo X que ya tiene *B;* por lo que A pasa a un estado de espera. Permanece en este estado hasta que *B* llega a su término (ya sea con COMMIT o ROLLBACK), esto es cuando el bloqueo de *B* queda liberado y *A* puede continuar. En ese punto, *A* ve un valor *confirmado* (ya sea el valor anterior a *B,* si *B* termina con una instrucción ROLLBACK, o en caso contrario, el nuevo valor después de *B).* De cualquier forma, *A* ya no depende de una actualización no confirmada.

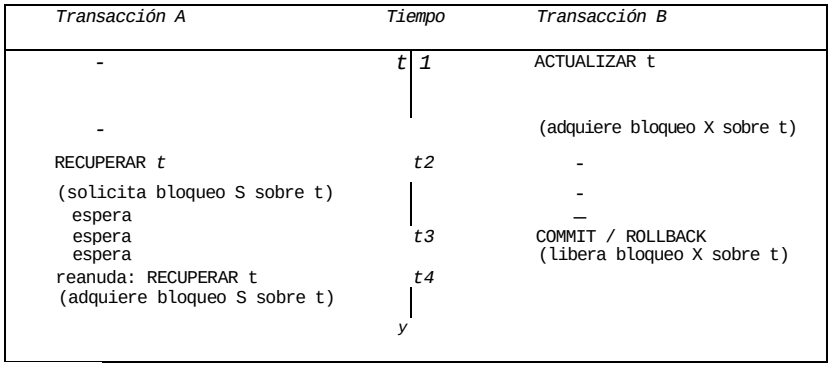


Figura 15.7 La transacción A no puede ver un cambio no confirmado en el tiempo f2.

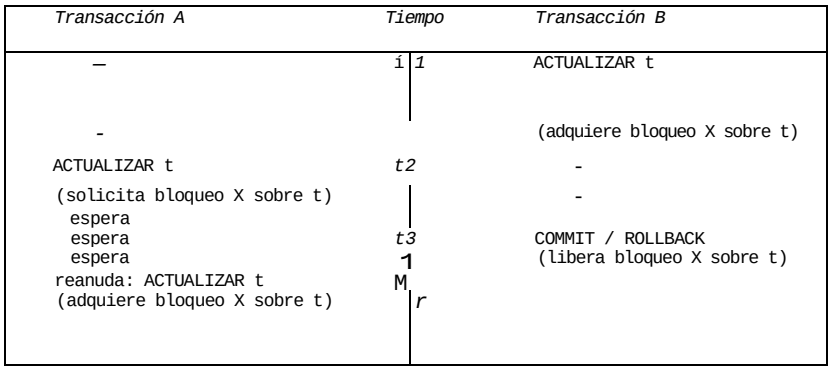


Figura 15.8 La transacción *A* no puede actualizar un cambio no confirmado en el tiempo í2.

El problema del análisis inconsistente

La figura 15.9 es una versión modificada de la figura 15.4, que muestra lo que sucedería en la ejecución intercalada de esa figura bajo el protocolo de bloqueo de la sección 15.3. La actualización de la transacción *B* en el tiempo *t6* no es aceptada debido a que es una petición implícita de un bloqueo X sobre ACC 1, y tal petición entra en conflicto con el bloqueo S que ya tiene A; por lo tanto, *B* pasa a un estado de espera. De forma similar, la recuperación de la transacción *A* en el tiempo *t7* tampoco es aceptada debido a que es una petición implícita para un bloqueo S sobre ACC 3, y tal petición entra en conflicto con el bloqueo X que ya tiene *B;* por lo tanto, A también pasa a un estado de espera. Entonces, nuevamente el bloqueo resuelve el problema original (el problema de análisis inconsistente, en este caso) forzando un bloqueo mortal. Nuevamente vea la sección 15.5.

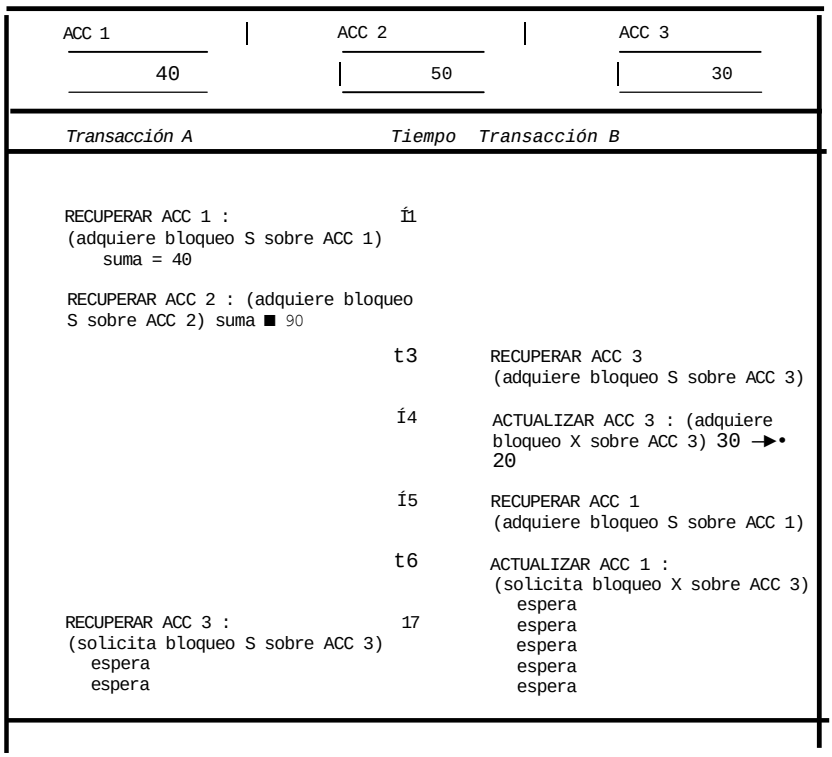


Figura 15.9 Se impide el análisis inconsistente, pero ocurre un bloqueo mortal en el tiempo í7.

5.5 BLOQUEO MORTAL

Ahora ya hemos visto la forma en que podemos usar el bloqueo para resolver los tres problemas básicos de la concurrencia. Sin embargo, desafortunadamente también hemos visto que el bloqueo puede introducir problemas por sí solo, principalmente el problema del **bloqueo mortal.** En la sección anterior proporcioné dos ejemplos de bloqueo mortal. La figura 15.10 muestra una versión un poco más general del problema, en donde *rl* y *r2 (r* de "recurso") pretenden representar cualquier objeto bloqueable, no solamente tupias de base de datos (vea la sección 15.8), y las instrucciones "BLOQUEO...EXCLUSIVO" representan cualquier operación que adquiera bloqueos (exclusivos), ya sea explícita o implícitamente.

El bloqueo mortal es una situación en la que dos o más transacciones se encuentran en estados simultáneos de espera, cada una de ellas esperando que alguna de las demás libere un bloqueo para poder continuar.* La figura 15.10 muestra un bloqueo mortal que involucra a dos

*A1 bloqueo mortal también se le conoce, coloquialmente, como *abrazo mortal.*

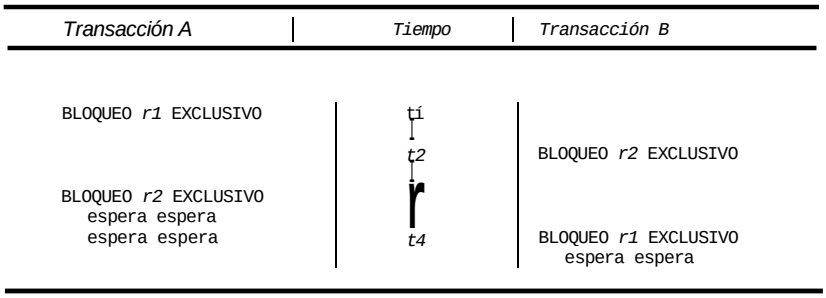


Figura 15.10 Un ejemplo de bloqueo mortal.

transacciones, pero también son posibles, al menos en principio, bloqueos mortales que involucren a tres, cuatro, o más transacciones. Sin embargo, queremos hacer notar que los experimentos con el System R mostraron aparentemente que en la práctica los bloqueos mortales casi nunca involucran a más de dos transacciones [15.8].

Si ocurre un bloqueo mortal es preferible que el sistema lo detecte y lo rompa. La detección del bloqueo mortal implica la detección de un ciclo en el **grafo de espera** (es decir, el grafo de "quién está esperando a quién", vea el ejercicio 15.4). La ruptura del bloqueo mortal implica seleccionar una de las transacciones bloqueadas mortalmente —es decir, una de las transacciones dentro del ciclo del grafo— como **víctima** y entonces deshacerla liberando por lo tanto sus bloqueos y permitiendo que continúen las demás transacciones. *Nota:* En la práctica no todos los sistemas detectan los bloqueos mortales, sino que algunos usan un mecanismo de tiempo y asumen simplemente que una transacción que no ha realizado algún trabajo durante cierto periodo preestablecido, está bloqueada mortalmente.

De paso, observe que la víctima ha "fallado" y ha sido deshecha *sin ser culpable.* Algunos sistemas volverán a iniciar esta transacción desde el principio, bajo la suposición de que las condiciones que causaron el bloqueo mortal probablemente ya no existirán. Otros sistemas simplemente regresan un código de excepción "víctima de bloqueo mortal" hacia la aplicación, y es asunto del programa manejar la situación de forma adecuada. El primero de estos dos enfoques es preferible desde el punto de vista del programador. Pero incluso si el programador tiene que involucrarse de vez en cuando, *siempre* es preferible ocultar el problema ante el usuario final, por razones obvias.

15.6 SERIABILIDAD

Ya hemos puesto las bases para explicar la noción fundamental de **seriabilidad.** La seriabilidad es el **criterio de corrección** aceptado comúnmente para la ejecución de un conjunto dado de transacciones. Para ser más precisos, se considera que la ejecución de un conjunto dado de transacciones es correcta cuando es *seriable;* es decir, cuando produce el mismo resultado que una ejecución serial de las mismas transacciones, ejecutando una a la vez. Ésta es la justificación de esta aseveración:

1. Las transacciones individuales son tomadas como correctas; es decir, se da por hecho que transforman un estado correcto de la base de datos en otro estado correcto.
2. Por lo tanto también es correcta la ejecución de una transacción a la vez en cualquier orden serial, y se dice "cualquier" orden serial debido a que las transacciones individuales son consideradas independientes entre sí.
3. Por lo tanto, una ejecución *intercalada* es correcta cuando equivale a alguna ejecución se rial; es decir, cuando es seriable.

Regresando a los ejemplos de la sección 15.2 (figuras 15.1 a 15.4), podemos ver que el problema en cada caso fue que la ejecución intercalada *no* era seriable; es decir, la ejecución intercalada nunca fue equivalente a ejecutar *A* después de *B,* o bien *B* después de *A.* Y el efecto del esquema de bloqueo tratado en la sección 15.3 fue precisamente/orzar la seriabilidad en cada uno de los casos. En las figuras 15.7 y 15.8, la ejecución intercalada fue equivalente *&A* después de *B.* En las figuras 15.6 y 15.9 ocurrió un bloqueo mortal, lo que implica que alguna de las dos transacciones debería ser deshecha y, supuestamente, ser ejecutada más tarde. Si *A* es la que se deshace, entonces la ejecución intercalada nuevamente se convierte en el equivalente *átA* después de *B.*

Terminología: Dado un conjunto de transacciones, a cualquier ejecución de esas transacciones, intercaladas o no, se le llama **plan.** La ejecución de una transacción a la vez, sin intercalado, constituye un plan **serial;** mientras que un plan que no sea serial es **intercalado** (o simplemente no serial). Se dice que dos planes son **equivalentes** cuando garantizan que producirán el mismo resultado independientemente del estado inicial de la base de datos. Por lo tanto, un plan es correcto (es decir, seriable) cuando equivale a un plan serial.

El punto que vale la pena enfatizar es que dos planes seriales diferentes que involucran el mismo conjunto de transacciones bien pueden producir resultados diferentes y, por lo tanto, dos planes intercalados diferentes que involucran a esas transacciones, también pueden producir resultados diferentes, aunque ambos sean considerados como correctos. Por ejemplo, supongamos que la transacción *A* es de la forma "sumar 1 a *x"* y la transacción *B* es de la forma "duplicar *x"* (donde *x* es algún elemento de la base de datos). Supongamos también que el valor inicial de *x* es 10. Entonces el plan serial *B* después de *A,* dax = 22; mientas que el plan serial *A* después de *B,* da* = 21. Estos dos resultados son igualmente correctos y cualquier plan que garantice que sea equivalente a *B* después de *A* o bien a A después de *B,* también es correcto. Vea el ejercicio 15.3 al final del capítulo.

El concepto de seriabilidad fue presentado por primera vez (aunque no con ese nombre) por Eswaran *et al.* en la referencia [15.5]. El mismo artículo también proporciona un teorema importante, llamado el **teorema de bloqueo de dos fases,** que definimos brevemente de la siguiente forma:*

> Si todas las transacciones obedecen el "protocolo de bloqueo de dos fases", entonces todos los planes intercalados posibles son seriables.

El **protocolo de bloqueo de dos fases** tiene la siguiente forma:

1. Antes de operar sobre cualquier objeto (por ejemplo, una tupia de base de datos), una transacción debe adquirir un bloqueo sobre ese objeto.
2. Después de liberar un bloqueo, una transacción nunca deberá adquirir ningún otro bloqueo.

---

* El bloqueo de dos fases no tiene nada que ver con la confirmación de dos fases, simplemente tienen nombres similares.

Por lo tanto, una transacción que obedece este protocolo tiene dos fases: una fase de adquisición de bloqueo, o fase de "crecimiento", y una fase de liberación de bloqueo, o de "reducción". *Nota:* En la práctica, la fase de reducción a menudo está comprimida en la operación única COMMIT (o ROLLBACK) al final de la transacción. De hecho, el protocolo de acceso a datos que tratamos en la sección 15.3 puede ser visto como una forma robusta del protocolo de bloqueo de dos fases.

La noción de seriabilidad es de gran ayuda para entender con claridad esta área potencialmente confusa; aquí proporcionamos algunas observaciones adicionales sobre ello. Supongamos que / es un plan intercalado que involucra un conjunto de transacciones 77, 72, ..., *Tn.* Si / es seriable, existe entonces un plan serial *S* que involucra a *TI, T2, ...,Tn* de manera que / es equivalente a *S.* Se dice que *S* es una **seriación** de /. Como ya hemos visto, *S* no es necesariamente único; es decir, un plan intercalado puede tener más de una seriación.

Dejemos ahora que *Ti* y *Tj* sean dos transacciones cualesquiera en el conjunto *TI,* 72,..., *Tn.* Supongamos, sin perder la generalidad, que *Ti* precede a *Tj* en la seriación *S.* Por lo tanto, en el plan intercalado / el efecto debe ser como si *Ti* en realidad se ejecutara antes que *Tj.* En otras palabras, una caracterización informal pero muy útil de la seriabilidad es que si *A* y *B* son dos transacciones cualesquiera involucradas en algún plan seriable, entonces *A* precede lógicamente a *B* o *B* precede lógicamente a A en ese plan; esto es, ***B* puede ver la salida de A o A puede ver la salida de*B.*** (Si A actualiza los recursos r,s .....(yfivea cualquiera de estos recursos como entrada, entonces *B* ve a todos ellos *ya sea* como quedan después de haber sido actualizados por A *o* como estaban antes de haber sido actualizados por A, pero no una mezcla de los dos.) En forma inversa, si el efecto no es como si A se ejecutara antes que *B* o *B* antes que *A,* entonces el plan no es seriable y no es correcto.

En conclusión, el punto importante es que si una transacción A no es de dos fases (es decir, no obedece el protocolo de bloqueo de dos fases) entonces *siempre* es posible construir alguna otra transacción *B* que pueda ejecutarse intercalada con A, en forma tal que produzca un plan general que no sea seriable ni correcto. Ahora bien, si nos interesa reducir la disputa de recursos, y por consiguiente mejorar el rendimiento total, los sistemas reales comúnmente permiten construir transacciones que no son de dos fases; es decir, transacciones que "liberan tempranamente los bloqueos" (antes del COMMIT) y luego continúan adquiriendo más bloqueos. Sin embargo, debe quedar claro que tales transacciones representan una propuesta riesgosa, ya que permitir que una transacción A dada no sea de dos fases equivale a apostar que una transacción *B* que la interfiere nunca llegará a existir en el sistema junto con A (ya que si esto ocurre, el sistema tendería a producir respuestas erróneas).

15.7 NIVELES DE AISLAMIENTO

Utilizamos el término **nivel de aislamiento** para referirnos a lo que podría ser descrito vagamente como el *grado de interferencia* que una transacción dada es capaz de tolerar por parte de las transacciones concurrentes. Entonces, si queremos garantizar la seriabilidad, ¡no podemos aceptar ninguna cantidad de interferencia!; en otras palabras, el nivel de aislamiento debe ser el máximo posible. Sin embargo, como indiqué al final de la sección anterior, los sistemas reales comúnmente permiten que las transacciones operen a niveles de aislamiento que son menores a este máximo, por una diversidad de razones pragmáticas.

Nota: Como lo sugiere el párrafo anterior, el nivel de aislamiento es visto generalmente como una propiedad de las *transacciones.* De hecho, no hay razón por la cual una transacción no deba operar simultáneamente a niveles diferentes de aislamiento en distintas partes de la base de datos, al menos en principio. Sin embargo, por simplicidad, continuaremos pensando en el nivel de aislamiento como una propiedad de toda la transacción.

Podemos definir al menos, cinco niveles de aislamiento, pero la referencia [15.9] y el SQL estándar definen solamente cuatro, mientras que el DB2 sólo soporta dos actualmente. En términos generales, entre mayor sea el nivel de aislamiento, menor será la interferencia (y menor la concurrencia); y entre más bajo sea el nivel de aislamiento, mayor será la interferencia (y mayor la concurrencia). Como ejemplo consideraremos los dos niveles soportados por DB2, que son llamados **estabilidad de cursor y lectura repetible,** respectivamente. La *lectura repetible* (RR) es el nivel máximo. Si todas las transacciones operan a este nivel, todos los planes son seriables (las explicaciones en las secciones 15.3 y 15.4 supusieron tácitamente este nivel de aislamiento). Por el contrario, bajo la *estabilidad de cursor* (CS), si una transacción *TI*

- Obtiene direccionabilidad hacia alguna tupia *t** y luego;
- Adquiere un bloqueo sobre *t,* y luego;
- Abandona su direccionabilidad hacia *t* sin actualizarla y después;
- No promueve su bloqueo hacia el nivel X, entonces;
- Ese bloqueo puede ser liberado sin tener que esperar hasta el final de la transacción.

Pero observe que ahora alguna otra transacción 72 puede actualizar a *t* y confirmar el cambio. Si la transacción 77 regresa y vuelve a ver a *t,* verá el cambio y por lo tanto es posible que (en efecto) vea un estado inconsistente de la base de datos. Por el contrario, bajo la lectura repetible (RR), *todos* los bloqueos de tupias (no sólo los bloqueos X) se mantienen hasta el final de la transacción, por lo que no puede ocurrir el problema que aquí mencionamos. Se presentan los siguientes puntos:

1. El problema anterior *no* es el único que puede ocurrir bajo CS, sino que simplemente es el más fácil de explicar. Pero desgraciadamente sugiere que RR sólo es necesario en el caso, comparativamente poco probable, de que una transacción dada necesite ver la misma tupia dos veces. Por el contrario, hay argumentos que sugieren que RR *siempre* es una mejor al ternativa que CS, ya que una transacción ejecutada bajo CS no es de dos fases y por lo tanto (como expliqué en la sección anterior) no puede garantizar la seriabilidad. Por supuesto, el argumento en contra es que CS da más concurrencia que RR (probablemente, pero no nece sariamente).
2. Una implementación que soporta cualquier nivel de aislamiento menor que el máximo, pro porcionará normalmente algunas facilidades de control de concurrencia explícitas (por lo general, instrucciones LOCK explícitas) para permitir que los usuarios escriban sus aplica ciones en forma tal que garanticen la seguridad en ausencia de una garantía otorgada por el propio sistema. Por ejemplo, DB2 proporciona una instrucción LOCK TABLE explícita que

---

* Hace esto poniendo un *cursor* que apunte hacia la tupia, como expliqué en el capítulo 4 (de ahí el nombre "estabilidad de cursor"). Para ser más precisos, debemos mencionar que el bloqueo que adquiere 77 sobre / en DB2 es un bloqueo de "actualización" (U) y no un bloqueo S (vea la referencia [4.20]).

permite a los usuarios que operan al nivel CS adquirir bloqueos explícitos por encima de los que DB2 adquiere automáticamente para hacer cumplir ese nivel. (Sin embargo, observe que SQL estándar no incluye mecanismos de control de concurrencia explícitos, vea la sección 15.9.)

Para finalizar esta sección, observemos que la caracterización anterior de RR como el nivel máximo de aislamiento se refiere a la lectura repetible tal como está implementada en DB2. Desafortunadamente, SQL estándar utiliza el mismo término "lectura repetible" para representar un nivel de aislamiento que es estrictamente menor que el nivel máximo (nuevamente vea la sección 15.9).

15.8 BLOQUEO POR APROXIMACIÓN

Hasta este punto, hemos estado dando por hecho que la unidad para efectos de bloqueo es la tupia individual. Sin embargo, en principio no hay razón para no aplicar los bloqueos a unidades de datos más grandes o más pequeñas: toda una variable de relación (varrel) e incluso toda una base de datos; o si nos vamos al extremo opuesto, un componente específico dentro de una tupia en particular. Hablamos de la **granularidad del bloqueo** [15.9 y 15.10]. Como es usual hay un compromiso: entre más fina sea la granularidad mayor será la concurrencia; entre más burda, se necesitarán establecer y probar menos bloqueos y será menor la sobrecarga. Por ejemplo, si una transacción tiene un bloqueo X sobre una varrel, no hay necesidad de poner bloqueos X sobre tupias individuales dentro de esa varrel. Por otra parte, ninguna transacción concurrente podrá obtener un bloqueo sobre esa varrel ni sobre tupias que estén dentro de la misma.

Supongamos que una transacción *T* solicita un bloqueo X sobre alguna varrel *R.* Al recibir la petición de *T,* el sistema debe poder decir si alguna otra transacción ya tiene un bloqueo sobre cualquier tupia de *R;* porque de ser así, la petición de *T* no podrá ser otorgada en este momento. ¿Cómo puede el sistema detectar este conflicto? Obviamente, no es conveniente que tengamos que analizar cada una de las tupias de *R* para ver si alguna de ellas está bloqueada por otra transacción, ni tener que analizar cada uno de los bloqueos para ver si alguno de ellos es para una tupia de *R.* En su lugar, presentamos el **protocolo de bloqueo por aproximación,** de acuerdo con el cual ninguna transacción puede adquirir un bloqueo sobre una tupia sin adquirir antes un bloqueo —probablemente un bloqueo por *aproximación* (vea más adelante)— sobre la varrel que lo contiene. La detección de conflictos en el ejemplo se convierte entonces en un asunto relativamente simple, ya que sólo hay que ver si alguna transacción tiene un bloqueo conflictivo *a nivel varrel.*

Ya hemos indicado que los bloqueos X y S tienen sentido para todas las varrels, así como para las tupias individuales. De acuerdo con las referencias [15.9 y 15.10], ahora presentamos tres tipos nuevos de bloqueos llamados **bloqueos por aproximación,** que también tienen sentido para las varrels pero no para las tupias individuales. Los nuevos tipos de bloqueos se llaman bloqueo de **aproximación compartida (IS),** de **aproximación exclusiva (IX)** y de **aproximación compartida exclusiva** (SIX), respectivamente. Pueden ser definidos informalmente de la siguiente forma (suponiendo que la transacción *T* ha solicitado un bloqueo del tipo indicado sobre la varrel *R;* para que la definición esté completa, también incluimos definiciones para los tipos X y S):

- **IS:** *T* pretende poner bloqueos S sobre tupias individuales de *R* para garantizar su estabili dad mientras están en proceso.
- IX: Igual que IS, pero *además T* puede actualizar tupias individuales en *R* y por lo tanto pondrá bloqueos X sobre esas tupias.
- S: *T* puede tolerar lectores concurrentes, pero no actualizadores concurrentes en *R. T* por sí misma no actualizará ninguna tupia en *R.*
- **SIX:** Combina a S e IX; es decir, *T* puede tolerar lectores concurrentes, pero no actuali zadores concurrentes en *R. Además T* puede actualizar tupias individuales en *R* y por lo tanto pondrá bloqueos X sobre esas tupias.
- X: *T* no puede tolerar ningún acceso concurrente a *R; y T* mismo puede actualizar o no tu pias individuales en *R.*

Las definiciones formales de estos cinco tipos de bloqueo están dadas por una versión extendida de la matriz de compatibilidad de tipos de bloqueo que tratamos anteriormente en la sección 15.3. Vea la figura 15.11.

	X	SIX	IX	S	IS	-
X	No	NO	No	NO	NO	Sí
SIX	No	No	NO	No	Sí	Sí
IX	No	No	Sí	No	Sí	Sí
S	NO	NO	No	Sí	Sí	Sí
IS	No	Sí	Sí	Sí	Sí	Sí
-	Sí	Sí	Sí	Sí	Sí	Sí

Figura 15.11 Matriz de compatibilidad extendida para incluir los bloqueos por aproximación.

Ahora presento un enunciado más preciso del protocolo de bloqueo por aproximación:

1. Antes de que una transacción dada pueda adquirir un bloqueo S sobre un tupia dada, primero debe adquirir un bloqueo IS o uno más fuerte sobre la varrel que contiene a esa tupia (como se muestra a continuación).
2. Antes de que una transacción dada pueda adquirir un bloqueo X sobre un tupia dada, primero debe adquirir un bloqueo IX o uno más fuerte sobre la varrel que contiene a esa tupia (como se muestra a continuación).

(Sin embargo, observe que todavía no es una definición completa. Vea el comentario de la referencia [15.9].)

La noción de la *fuerza relativa del bloqueo,* mencionada en el protocolo anterior, puede ser explicada de la siguiente forma. Consulte el **grafo de precedencia** de la figura 15.12. Decimos que el tipo de bloqueo *L2* es más fuerte —es decir, está más arriba en el grafo— que el bloqueo de tipo *Ll* si, y sólo si, cada vez que haya un "No" (conflicto) en la columna *Ll* de la matriz de compatibilidad para una fila dada, también hay un "No" en la columna *L2* para esa misma fila

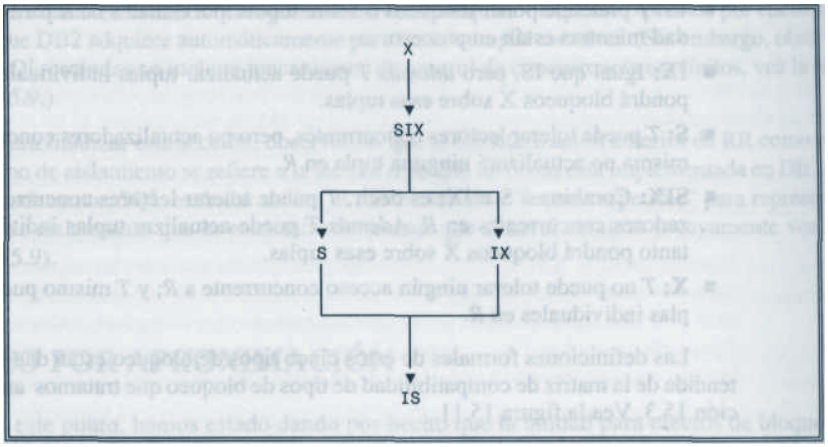


Figura 15.12 Grafo de precedencia de los tipos de bloqueo.

(vea la figura 15.11). Observe que una petición de bloqueo que falla para un tipo de bloqueo dado ciertamente fallará para un tipo de bloqueo más fuerte (y este hecho implica que siempre es más seguro usar un tipo de bloqueo más fuerte que el estrictamente necesario). Observe también que ni S ni IX son más fuertes que el otro.

Vale la pena observar que en la práctica los bloqueos a nivel varrel requeridos por el protocolo de bloqueo por aproximación, generalmente son adquiridos implícitamente. Por ejemplo, para una transacción de sólo lectura, el sistema probablemente adquirirá un bloqueo IS implícito sobre cada varrel a la que la transacción acceda. Por el contrario, para una transacción de actualización probablemente adquirirá bloqueos IX. Pero el sistema probablemente tendrá que proporcionar también una instrucción LOCK explícita de algún tipo para permitir que las transacciones adquieran bloqueos S, X o SIX a nivel varrel cuando lo deseen. Por ejemplo, esta instrucción es soportada por DB2 (aunque sólo para los bloqueos S y X, y no para los SIX).

Cerramos esta sección con un comentario sobre el **escalamiento de bloqueos,** que está implementado en muchos sistemas y que representa un intento para equilibrar los requerimientos conflictivos entre una concurrencia alta y una sobrecarga baja en la administración de bloqueos. La idea básica es que cuando el sistema llega a algún punto predefinido, reemplaza automáticamente un conjunto de bloqueos de granularidad fina por un solo bloqueo de granularidad burda (eliminando, por ejemplo, un conjunto de bloqueos S a nivel de tupias individuales y convirtiendo el bloqueo IS de la varrel que los contiene en un bloqueo S). Esta técnica parece funcionar bien en la práctica.

15.9 PROPIEDADES DE SQL

El estándar de SQL no proporciona ninguna posibilidad de bloqueo explícito (de hecho, no menciona para nada al bloqueo como tal). Sin embargo, requiere que la implementación proporcione las garantías usuales relativas a la interferencia, o a su ausencia, entre transacciones que se eje-

cutan concurrentemente. En particular, requiere que las actualizaciones hechas por una transacción 77 no sean visibles ante ninguna transacción 72 distinta hasta que la transacción 77 termine con una confirmación. Terminar con una confirmación ocasiona que todas las actualizaciones hechas por la transacción se tornen visibles ante las demás transacciones. Terminar con una instrucción ROLLBACK ocasiona que todas las actualizaciones hechas por la transacción sean canceladas.

Nota: Lo anterior supone que todas las transacciones se ejecutan con un nivel de aislamiento READ COMMITTED, REPEATABLE READ o SERIALIZABLE (vea la parte que viene a continuación). Se aplican consideraciones especiales para las transacciones ejecutadas al nivel READ UNCOMMITTED; esto se debe a que (a) se les permite realizar "lecturas sucias" —vea nuevamente la siguiente parte— pero (b) deben estar definidas como READ ONLY (como indiqué en el capítulo **14).**

Niveles de aislamiento

Recuerde del capítulo 14 que SQL incluye una instrucción llamada SET TRANSACTION, la cual utilizamos para definir determinadas características de la siguiente transacción que va a ser iniciada. Una de estas características es el *nivel de aislamiento.* Los niveles posibles son **READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ** o **SERIALIZABLE.*** El nivel predeterminado es SERIALIZABLE, ya que si se especifica alguno de los otros tres, la implementación tiene la libertad de asignar algún nivel superior (en donde "superior" está definido en términos del ordenamiento SERIALIZABLE > REPEATABLE READ > READ COMMITTED > READ UNCOMMITTED).

Si todas las transacciones son ejecutadas al nivel de aislamiento SERIALIZABLE (el predeterminado) queda garantizado que la ejecución intercalada de cualquier conjunto de transacciones concurrentes es seriable. Sin embargo, si alguna transacción se ejecuta a un nivel de aislamiento menor, la seriabilidad podría ser violada de varias formas diferentes. El estándar define tres formas específicas en las que se puede violar la seriabilidad: *lectura sucia, lectura no repetible y fantasmas:*

- **Lectura sucia.** Supongamos que la transacción *TI* realiza una actualización sobre alguna fila, luego la transacción *T2* recupera esa fila y la transacción *TI* termina con una instruc ción deshacer. Entonces, la transacción *T2* ha visto una fila que ya no existe, y que en cierto sentido nunca existió (debido a que la transacción *TI* efectivamente nunca fue ejecutada).
- **Lectura no repetible.** Supongamos que la transacción *TI* recupera una fila, luego la tran sacción *T2* actualiza esa fila y después la transacción *TI* recupera nuevamente la "misma" fila. La transacción *TI* ha recuperado la "misma" fila dos veces, pero ve dos valores diferen tes en ella.
- **Fantasmas.** Supongamos que la transacción *TI* recupera el conjunto de todas las filas que satisfacen alguna condición (por ejemplo, todas las filas de proveedores que satisfacen la condición de que la ciudad es París). Supongamos que la transacción 72 inserta entonces una nueva fila que satisface la misma condición. Si la transacción *TI* repite ahora su peti ción de recuperación, verá una fila que antes no existía, un "fantasma".

---

* SERIALIZABLE no resulta una buena palabra reservada en este caso, ya que se supone que los *planes* son los seriables y no las *transacciones.* Un mejor término podría ser simplemente TWO PHASE, que significa que la transacción obedecerá (o será obligada a obedecer) el protocolo de bloqueo de dos fases.

Los diferentes niveles de aislamiento están definidos en términos de las violaciones de la seriabilidad que permiten. Están resumidos en la figura 15.13 ("Sí" significa que puede ocurrir la violación indicada, y "No" que no puede ocurrir).

Nivel de aislamiento	*Lectura sucia*	*Lectura no repetible*	*Fantasma*
READ UNCOMMITTED	Sí	Si	Sí
READ COMMITTED	No	Si	Sí
REPEATABLE READ	No	No	Sí
SERIALIZABLE	No	No	No

Figura 15.13 Niveles de aislamiento de SQL.

Una pregunta obvia es: ¿Cómo puede el sistema evitar los "fantasmas"? La respuesta es que debe bloquear la *ruta de acceso* que utiliza para obtener los datos en consideración. En el ejemplo anterior (relacionado con los proveedores de París), si la ruta de acceso fuera un índice de las ciudades de los proveedores, entonces el sistema debería bloquear la entrada de París en ese índice. Tal bloqueo evitaría la creación de fantasmas, puesto que esta creación requeriría la actualización de la ruta de acceso (la entrada del índice en el ejemplo). Vea la referencia [14.12] para comentarios adicionales.

Concluimos esta sección remarcando el hecho de que REPEATABLE READ del SQL y la "lectura repetible" (RR) de DB2, no representan al mismo nivel de aislamiento. De hecho, el RR de DB2 es el mismo que el SERIALIZABLE del SQL.

15.10 RESUMEN

Hemos analizado el tema del **control de la concurrencia.** Comenzamos viendo los tres problemas que pueden presentarse en la ejecución intercalada de transacciones concurrentes cuando no hay tal control: el problema de la **actualización perdida,** el problema de la **dependencia no confirmada** y el problema del **análisis inconsistente.** Todos estos problemas se presentan en planes que no son **seriables;** es decir, que no son equivalentes a algún plan serial que involucre las mismas transacciones.

La técnica más ampliamente utilizada para el manejo de estos problemas es el **bloqueo.** Exis ten dos mecanismos básicos de implementación del bloqueo, el **bloqueo compartido (S) y < bloqueo exclusivo** (X). Si una transacción tiene un bloqueo S sobre un objeto, las demás transacciones también pueden adquirir un bloqueo S sobre ese objeto, pero no pueden adquirir un bloqueo X; si una transacción tiene un bloqueo X sobre un objeto, ninguna otra puede adquirir un bloqueo de ningún tipo sobre ese objeto. Después presentamos un protocolo para el uso de esto bloqueos a fin de asegurar que los problemas de la actualización perdida (y los demás) no se produzcan: la adquisición de un bloqueo S sobre todo lo recuperado, la adquisición de un bloqueo X sobre todo lo actualizado y la conservación de todos los bloqueos hasta el final de la transacción. Este protocolo hace cumplir la seriabilidad.

El protocolo que acabo de describir es una forma fuerte (pero común) del **protocolo de bloqueo de dos fases.** Es posible demostrar que si todas las transacciones obedecen este protocolo, entonces todos los planes son seriables. Este es el **teorema del bloqueo de dos fases.** Un plan seriable implica que si *A* y *B* son dos transacciones cualesquiera involucradas en ese plan, *A*

puede ver la salida de *B* o *B* puede ver la salida de *A.* Desafortunadamente, el protocolo de bloqueo de dos fases puede derivar en **bloqueos mortales.** Los bloqueos mortales son resueltos eligiendo una de las transacciones bloqueadas mortalmente como **víctima** y deshaciéndola (liberando por lo tanto a todos sus bloqueos).

En general, cualquier cosa menor que la seriabilidad completa no puede garantizar ser segura. Sin embargo, los sistemas normalmente permiten que las transacciones operen a un **nivel de aislamiento** que es inseguro, con el propósito de reducir la disputa por los recursos y de incrementar el rendimiento total de las transacciones. Describimos uno de estos niveles "no seguros", en la **estabilidad de cursor** (éste es el término de DB2; el término de SQL es READ COMMITTED).

Después consideramos brevemente la cuestión de la **granularidad del bloqueo** y la idea asociada del **bloqueo por aproximación.** Básicamente, antes de que una transacción pueda adquirir un bloqueo de cualquier tipo sobre algún objeto (digamos una tupia de base de datos), primero debe adquirir un bloqueo por aproximación adecuado al menos sobre el "padre" de ese objeto (es decir, la varrel que lo contiene, en caso de una tupia). En la práctica, tales bloqueos por aproximación por lo general se adquirirán implícitamente, en forma similar a como se adquieren implícitamente los bloqueos S y X sobre las tupias. Sin embargo, es necesario proporcionar algún tipo de **instrucciones LOCK explícitas** para permitir que una transacción adquiera bloqueos más fuertes (cuando los necesite) que los adquiridos implícitamente.

Por último, describimos el soporte para el control de la concurrencia en SQL. Básicamente, SQL no proporciona ninguna capacidad de bloqueo explícito. Sin embargo, soporta varios niveles de aislamiento, **READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ y SERIALIZABLE,** que probablemente implementará el DBMS mediante bloqueos tras bambalinas.

Concluimos esta parte del libro con otro comentario breve sobre la importancia de la integridad. Cuando decimos superficialmente que la base de datos es **correcta,** lo que queremos decir es que no viola ninguna restricción de integridad conocida. Por lo tanto, un sistema que proporciona poco soporte de la integridad sólo tendrá una percepción muy débil de lo que significa que la base de datos sea "correcta". Tomando en cuenta lo que dijimos en el capítulo 14 de que la *recuperación* significa recuperarse hasta un "estado correcto" previamente conocido, y lo dicho en el presente capítulo de que la *concurrencia* —o mejor dicho, el *control* de la concurrencia— significa que un plan seriable transforma un "estado correcto" de la base de datos en otro "estado correcto", podemos ver que la integridad es en realidad una cuestión mucho más fundamental que la recuperación o la concurrencia. Además, la integridad es una cuestión que existe incluso en un sistema de un solo usuario.

EJERCICIOS

15.1 Defina la seriabilidad.

15.2 Defina:

a. El protocolo de bloqueo de dos fases;

b. El teorema de bloqueo de dos fases.

15.3 Definamos las transacciones *TI, T2* y *T3* para que realicen las siguientes operaciones:

TI: Sumar 1 a A
T2: Duplicar A
T3: Desplegar *A* en la pantalla y establecer *A* en 1

(en donde A es algún elemento de la base de datos).

a. Suponga que permite que las transacciones *Tl,T2yT3* sean ejecutadas en forma concurrente. Si A tiene un valor inicial de cero, ¿cuántos resultados correctos posibles existen? Enumérelos.

b. Supongamos que la estructura interna de *Tl,T2y T3* es la que se indica a continuación. Si las transacciones son ejecutadas *sin* ningún bloqueo, ¿cuántos planes posibles existen?

71	T2	13
R1: RECUPERAR *A* EN tí ; tí := *t1* + 1 ¡ U1: ACTUALIZAR *A* CON tí ;	R2: RECUPERAR *A* EN t2 ; t2 := t2 * 2 ; U2: ACTUALIZAR *A* CON *t2* ;	R3: RECUPERAR *A* EN *t3* ; desplegar t3 ; U3: ACTUALIZA *A* CON 1 ;

c. Con el valor inicial dado para A (cero), ¿existe algún plan intercalado que produzca un resultado "correcto" y aun así no sea seriable?

d. ¿Existe algún plan que sea seriable pero que no pueda producirse si las tres transacciones obe decen el protocolo de bloqueo de dos fases?

15.4 Lo siguiente representa la secuencia de eventos en un plan que involucra a las transacciones *TI, T2,..., T12.* A, *B,..., H* son elementos de la base de datos.

tiempo	*W*	(Ti)	RECUPERAR	*A*
tiempo	tí			
tiempo	t	(72)	RECUPERAR	*B*
	-	(Ti)	RECUPERAR	*C*
	_	(74)	RECUPERAR	*D*
		(75)	RECUPERAR	*A*
	-	*(Ti)*	RECUPERAR	*E*
		(72)	ACTUALIZAR	F
		(T3)	RECUPERAR	*F*
	_	*(Tq)*	RECUPERAR	*F*
	-	(75)	ACTUALIZAR	*A*
	-	(Ti)	COMMIT ;	
	-	(T6)	RECUPERAR	*A*
	-	(T5)	ROLLBACK ¡	
	-	(76)	RECUPERAR	*C*
	-	*(T6)*	ACTUALIZAR	*C*
		(T7)	RECUPERAR	*G*
		(78)	RECUPERAR	*H*
	-	(79)	RECUPERAR	*G*
		(79)	ACTUALIZAR	*G*
	_	(78)	RECUPERAR	*E*
	-	(T7)	COMMIT ;	
	-	*(T9)*	RECUPERAR	*H*
	-	(73)	RECUPERAR	*G*
	-	(710)	RECUPERAR	*A*
	_	(79)	ACTUALIZAR	*H*
	-	*(T6)*	COMMIT ;	
	-	(Til)	RECUPERAR	*C*
	_	*(T12)*	RECUPERAR	*D*
	-	*(T12)*	RECUPERAR	*C*
	-	(72)	ACTUALIZAR	*F*
		(7ÍÍ)	ACTUALIZAR	C
	-	(712)	RECUPERAR	*A*
	_	(710)	ACTUALIZAR	*A*
	-	(712)	ACTUALIZAR	*D*
	-	(74)	RECUPERAR	*G*
tiempo	*tn*			

Suponga que "RECUPERAR *R"* (si es satisfactorio) adquiere un bloqueo S sobre *R* y "ACTUALIZAR *R"* (si es satisfactorio) promueve ese bloqueo al nivel X. Suponga también que todos los bloqueos se mantienen hasta el fin de la transacción. ¿Existe algún bloqueo mortal en el tiempo *tril*

15.5 Considere nuevamente los problemas de concurrencia que se ilustran en las figuras 15.1 a 15.4. ¿Qué pasaría en cada caso si todas las transacciones estuvieran ejecutándose bajo el nivel de aisla miento CS en vez de RR?

15.6 Proporcione las definiciones formales e informales de los tipos de bloqueos X, S, IX, IS y SIX.

15.7 Defina la noción de la fuerza relativa de los bloqueos y dé el grafo de precedencia corres pondiente.

15.8 Defina el protocolo de bloqueo por aproximación. ¿Cuál es el propósito de ese protocolo?

15.9 SQL define tres problemas de concurrencia: *lectura sucia, lectura no repetible* y *fantasmas.* ¿Cómo se relacionan con los tres problemas de concurrencia identificados en la sección 15.2?

15.10 Esboce un mecanismo de implementación para los protocolos de control de concurrencia de versión múltiple que se describen brevemente en el comentario de la referencia [15.1].

DEFERENCIAS Y BIBLIOGRAFÍA

Además de las siguientes, vea también las referencias [14.2], [14.10] y (especialmente) [14.12] del capítulo 14.

15.1 R.Bayer, M. Heller y A. Reiser: "Parallelism and Recovery in Database Systems", *ACM TODS* 5, No. 2 (junio, 1980).

Como dije en el capítulo 14, las áreas de aplicación más recientes (como la ingeniería de hardware y de software) a menudo involucran requerimientos complejos de procesamiento para los cuales los controles clásicos de administración de transacciones, tal como se describen en el cuerpo de este capítulo y el anterior, no son muy adecuados. El problema básico es que las transacciones complejas pueden durar horas o días, en vez de unos cuantos milisegundos como sucede en los sistemas tradicionales. En consecuencia:

1. Deshacer una transacción hasta su inicio puede ocasionar la pérdida inaceptable de una gran cantidad de trabajo.
2. El uso del bloqueo convencional puede ocasionar retrasos inaceptablemente largos (en espera que se liberen los bloqueos).

El presente artículo es uno de los varios que tratan estos problemas (otros incluyen a las referencias [15.7], [15.11 a 15.13] y [15.17]). Propone una técnica de control de concurrencia conocida como **bloqueo de versión múltiple** (llamada también **lectura de versión múltiple,** implementada ahora en varios productos comerciales). La mayor ventaja de esta técnica es que las operaciones de lectura nunca tienen que esperar, ya que pueden operar simultáneamente cualquier cantidad de lectores y *un solo escritor* en el mismo objeto lógico. Para ser más específicos:

- Las lecturas nunca se retrasan;
- Las lecturas nunca retrasan a las actualizaciones;
- Nunca es necesario deshacer una transacción de sólo lectura;
- El bloqueo mortal sólo es posible entre transacciones de actualización.

Estas ventajas son en especial importantes en los sistemas distribuidos (vea el capítulo 20), en donde las actualizaciones pueden tardar mucho tiempo y las consultas de sólo lectura podrían retrasarse excesivamente (y *viceversa).* La idea básica es la siguiente:

- Si la transacción *T2* solicita leer un objeto sobre el cual la transacción *TI* tiene actualmente un acceso de actualización, a la transacción *T2* se le da acceso a una versión *previamente con firmada* de ese objeto. Tal versión debe existir en algún lugar del sistema (probablemente en la bitácora), para efectos de recuperación.
- Si la transacción 72 solicita la actualización de un objeto sobre el cual la transacción *TI* tiene ac ceso de lectura, la transacción 72 obtiene acceso a ese objeto, mientras que la transacción *TI* con serva el acceso a su propia versión del objeto (la cual es ahora, la versión anterior).
- Si la transacción 72 solicita la actualización de un objeto sobre el cual la transacción *TI* tiene acceso de actualización, la transacción 72 entra a un estado de espera* (por lo tanto, el bloqueo mortal y la reversión forzada todavía son posibles, como indiqué anteriormente).

Por supuesto, el enfoque incluye controles adecuados para asegurar que cada transacción siempre vea un estado consistente de la base de datos.

15.2 Hal Berenson *et al:* "A Critique of ANSI SQL Isolation Levels", Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif, (mayo, 1995).

Este artículo es una crítica al intento de SQL estándar para caracterizar los niveles de aislamiento en términos de las violaciones a la seriabilidad (vea la sección 15.9): "Las definiciones no logran caracterizar adecuadamente varios de los niveles de aislamiento comunes, incluyendo las implementaciones del bloqueo estándar de los niveles cubiertos." El artículo remarca, en especial, que el estándar falla en la prohibición de las **escrituras sucias** (definidas como la posibilidad de que dos transacciones *TI* y 72 puedan realizar una actualización sobre la misma fila antes de que cualquiera de ellas termine).

Al parecer, el estándar no prohibe explícitamente las escrituras sucias. Lo que en realidad dice es lo siguiente (cambiando un poco las palabras):

- "Se garantiza que la ejecución de transacciones concurrentes al nivel de aislamiento SERIALIZABLE será seriable." En otras palabras, si todas las transacciones operan al nivel de aisla miento SERIALIZABLE *es necesario* que la implementación prohiba las escrituras sucias, ya que éstas en efecto violan la seriabilidad.
- "Los cuatro niveles de aislamiento garantizan que... no se perderán las actualizaciones." Esta declaración simplemente es de buenas intenciones, ya que las definiciones de los cuatro nive les de aislamiento no proporcionan ninguna garantía por sí mismas. Sin embargo, indica que quienes definieron el estándar *pretendieron* prohibir las escrituras sucias.
- "Los cambios realizados por una transacción no pueden ser percibidos por otras transacciones [a excepción de las que tienen nivel de aislamiento READ UNCOMMITTED] hasta que ter mine la transacción original con una confirmación." La pregunta aquí es, ¿qué significa exac tamente *percibido*[1]*}* ¿Sería posible que una transacción actualizara una parte de los "datos sucios" sin "percibirlos"?

Nota: Los comentarios anteriores fueron tomados de la referencia [4.19].

15.3 Philip A. Bernstein y Nathan Goodman: "Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems", Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canadá (octubre, 1980).

Trata un conjunto de enfoques para el control de la concurrencia basados no en el bloqueo, sino en **marcas de tiempo.** La idea básica es que si una transacción *A* inicia su ejecución antes que la transacción *B,* entonces el sistema deberá comportarse como si *A* fuera ejecutada completamente antes de que *B* diera inicio (como en un plan serial genuino). Por lo tanto, *A* nunca podría

---

* En otras palabras, aún pueden ocurrir conflictos de actualización/actualización y suponemos que en este caso el bloqueo es utilizado para resolverlos. Tal vez, en vez de ello, podemos emplear otras técnicas (por ejemplo las marcas de tiempo [15.3]).

ver ninguna actualización de *B* y, en forma similar, *A* no podría actualizar nada que ya *B* haya visto. Podemos hacer cumplir tales controles de la siguiente forma. Para cualquier petición a la base de datos, el sistema compara la marca de tiempo de la transacción solicitante con la marca de tiempo de la transacción que ha recuperado o actualizado más recientemente la tupia solicitada. Si hay un conflicto, podemos simplemente volver a iniciar la transacción solicitante con una nueva marca de tiempo (como en los métodos llamados *optimistas* [15.14]).

Como sugiere el título del artículo, las marcas de tiempo fueron introducidas originalmente en el contexto de un sistema distribuido (en donde parecía que el bloqueo imponía sobrecargas intolerables, debido a los mensajes necesarios para probar y establecer bloqueos, entre otras cosas). Pero muy probablemente no son adecuadas en un sistema que no es distribuido. Además, existe mucho escepticismo de que también sean prácticas en sistemas distribuidos. Un problema obvio es que cada tupia debe llevar la marca de tiempo de la última transacción que la *recuperó* (así como la marca de tiempo de la última transacción que la actualizó), ¡lo cual implica que cada lectura se convierta en una escritura! De hecho, la referencia [14.12] menciona que los esquemas de marcas de tiempo en realidad sólo son un caso degenerado de los esquemas de control de concurrencia optimistas [15.14], que a su vez sufren sus propios problemas.

15.4 M. W. Blasgen, J. N. Gray, M. Mitoma y T. G. Price: "The Convoy Phenomenon", *ACM Ope rating Systems Review 13,* No. 2 (abril, 1979).

El **efecto convoy** es un problema que surge con bloqueos de mucho tráfico, como el bloqueo necesario para escribir un registro en la bitácora en sistemas con *planeación por prioridad*. *Nota:* Aquí la "planeación" se refiere al problema de asignar ciclos de máquina a las transacciones, no al intercalado de las operaciones sobre la base de datos que realizan diferentes transacciones (tal como es tratado en el capítulo).

El problema es el siguiente. Si una transacción *T* está usando un bloqueo de mucho tráfico y es suspendida por el planificador del sistema; es decir, es forzada hacia un estado de espera (tal vez porque se ha terminado su tiempo de ejecución), entonces se formará un *convoy* de transacciones donde todas estarán esperando su turno para usar ese bloqueo de mucho tráfico. Cuando Tsale de su estado de espera, no tarda mucho en liberar el bloqueo. Pero debido precisamente a que el bloqueo es de mucho tráfico, la misma *T* probablemente se volverá a unir al convoy antes de que la siguiente transacción haya terminado de utilizar el recurso; y por lo tanto no podrá continuar su procesamiento, cayendo nuevamente en un estado de espera.

En la mayoría de los casos (pero no en todos), la raíz del problema es que el planificador es parte del sistema operativo subyacente y no del DBMS, y por lo tanto está diseñado con base en suposiciones diferentes. Como indica el autor, una vez que el convoy queda establecido, tiende a ser estable; el sistema está en un estado de "bloqueo por basura" y la mayoría de los ciclos de máquina están dedicados a la conmutación de procesos por lo que no se realiza mucho trabajo útil. Una solución sugerida, además de la posibilidad de reemplazar al planificador, es otorgar el bloqueo en orden aleatorio en vez de atender en el orden: el primero que llega es el primero que sale.

15.5 K. P. Eswaran, J. N. Gray, R. A. Lorie e I. L. Traiger: "The Notions of Consistency and Predi cate Locks in a Data Base System", *CACM 19,* No. 11 (noviembre, 1976).

Es el artículo que puso por primera vez las bases teóricas del tema del control de la concurrencia.

15.6 Peter Franaszek y John T. Robinson: "Limitations on Concurrency in Transaction Processing", *ACM TODS 10,* No. 1 (marzo, 1985).

Vea el comentario de la referencia [15.14].

15.7 Peter A. Franaszek, John T. Robinson y Alexander Thomasian: "Concurrency Control for High Contention Environments", *ACM TODS 17,* No. 2 (junio, 1992).

Este artículo plantea que, por varias razones, es probable que los sistemas de procesamiento de transacciones futuros involucren un grado de concurrencia mucho mayor que los sistemas actuales y que, por lo tanto, es probable que haya una mayor contención de datos en ellos. Los

autores presentan entonces "varios conceptos de control de concurrencia [sin bloqueo] y técnicas de planeación de transacciones que son aplicables a los ambientes de gran contención" que supuestamente (con base en experimentos con modelos de simulación) "pueden proporcionar beneficios sustanciales" en tales ambientes.

15.8 J. N. Gray: "Experience with the System R Lock Manager", IBM San Jose Research Labora tory, memorando interno (primavera de 1980).

En realidad, esta referencia es sólo un conjunto de notas y no un artículo terminado, y sus hallazgos pueden ser un poco obsoletos actualmente. Sin embargo, contiene algunos comentarios interesantes, entre ellos:

- El bloqueo impone una sobrecarga cercana al diez por ciento en las transacciones en línea y cercana al uno por ciento en las transacciones por lotes.
- Es bueno dar soporte a diversas granularidades de bloqueo.
- El escalamiento automático de bloqueos funciona bien.
- Los bloqueos mortales son raros en la práctica y nunca involucran a más de dos transacciones. Casi todos los bloqueos mortales (97 por ciento) pueden ser evitados manejando los bloqueos U, como lo hace DB2 aunque no System R. (Los bloqueos U están definidos para ser com patibles con los bloqueos S pero no con otros bloqueos U, ni con los bloqueos X. Para mayo res detalles vea la referencia [4.20].)
- La lectura repetible (RR) es más eficiente y más segura que la estabilidad de cursor (CS).

15.9 J. N. Gray, R. A. Lorie y G. R. Putzolu: "Granularity of Locks in a Large Shared Data Base", Proc. 1st Int. Conf. on Very Large Data Bases, Framingham, Mass. (septiembre, 1975).

Es el artículo que presenta el concepto de **bloqueo por** aproximación. Como explico en la sección 15.8, el término "granularidad" se refiere al tamaño de los objetos que pueden ser bloqueados. Puesto que las distintas transacciones tienen obviamente diferentes características y requerimientos, es bueno que el sistema proporcione un rango de granularidades de bloqueo diferentes (como lo hacen muchos). Este artículo presenta un mecanismo de implementación para un sistema de granularidad múltiple basado en el bloqueo por aproximación.

Damos aquí más detalles sobre el **protocolo de bloqueo por aproximación,** ya que las explicaciones que presenté en el cuerpo del capítulo han sido simplificadas deliberadamente. En primer lugar, los tipos de objetos bloqueables no necesitan estar limitados a varrels y tupias, como dimos por hecho anteriormente. En segundo lugar, estos tipos de objetos bloqueables ni siquiera necesitan formar una jerarquía estricta, ya que la presencia de índices y otras estructuras de acceso significa que deben ser vistos más bien como un *grafo acíclico dirigido.* Por ejemplo, la base de datos de proveedores y partes puede contener (una forma almacenada de) la varrel P de partes así como un índice, digamos XP, sobre el atributo P#. Para obtener las tupias de la varrel P debemos comenzar con la base de datos en su conjunto, y luego ir ya sea directamente hacia la varrel y hacer un barrido secuencial o bien hacia el índice XP y a partir de él, hacia las tupias P requeridas. Por lo tanto, las tupias de P tienen dos "padres" en el grafo, P y XP, en donde ambos tienen a la base de datos como "padre".

Podemos ahora presentar el protocolo en su forma más general.

- La adquisición de un bloqueo X sobre un objeto dado adquiere implícitamente un bloqueo X sobre todos los hijos de ese objeto.
- La adquisición de un bloqueo S o SIX sobre un objeto dado adquiere implícitamente un blo queo S sobre todos los hijos de ese objeto.
- Antes de que una transacción pueda adquirir un bloqueo S o IS sobre un objeto dado, primero debe adquirir un bloqueo IS (o más fuerte) al menos sobre uno de los padres de ese objeto.
- Antes de que una transacción pueda adquirir un bloqueo X, IX o SIX sobre un objeto dado, primero debe adquirir un bloqueo IX (o más fuerte) sobre todos los padres de ese objeto.
- Antes de que una transacción pueda liberar un bloqueo sobre un objeto dado, primero debe liberar todos los bloqueos que tenga sobre los hijos de ese objeto.

En la práctica, el protocolo no impone tanta sobrecarga en el tiempo de ejecución como pudiéramos pensar, ya que en un momento dado la transacción podría tener la mayoría de los bloqueos que necesita. Por ejemplo, es probable que un bloqueo IX sea adquirido una sola vez sobre toda la base de datos en el momento de inicio del programa. Ese bloqueo se conservará a lo largo de la ejecución de todas las transacciones durante el tiempo de vida del programa.

15.10 J. N. Gray, R. A. Lorie, G. R. Putzolu e I. L. Traiger: "Granularity of Locks and Degrees of Con sistency in a Shared Data Base", en G. M. Nijssen (ed.), *Proc. 1FIP TC-2 Working Conf. on Modelling in Data Base Management Systems.* Amsterdam, Holanda: North-Holland/Nueva York, N.Y.; Elsevier Science (1976).

Es el artículo que presenta el concepto de nivel de aislamiento (bajo el nombre *grados de consistencia).*

15.11 Theo Harder y Kurt Rothermel: "Concurrency Control Issues in Nested Transactions", *The VLDB Journal* 2, No. 1 (enero, 1993).

Como expliqué en el capítulo 14, varios escritores han sugerido la idea de **transacciones anidadas.** Este artículo propone un conjunto adecuado de protocolos de bloqueo para tales transacciones.

15.12 J. R. Jordan, J. Banerjee y R. B. Batman: "Precision Locks", Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich, (abril-mayo, 1981).

El bloqueo de precisión es un esquema de bloqueo a nivel de tupia que garantiza que sólo serán bloqueadas aquellas tupias que lo necesitan (para lograr la seriabilidad), incluyendo los fantasmas. Es de hecho una forma de lo que en otros lugares se llama bloqueo por *predicado* [15.5]. Esto funciona (a) revisando las peticiones de actualización para ver si una tupia que va a ser insertada o borrada satisface una petición de recuperación anterior hecha por alguna transacción concurrente y (b) revisando las peticiones de recuperación para ver si una tupia que ya ha sido insertada o borrada por alguna transacción concurrente satisface la petición de recuperación en cuestión. Este esquema no es sólo bastante elegante, sino que los autores dicen que se comporta mejor que las técnicas convencionales (que por lo general bloquean mucho más).

15.13 Henry F. Korth y Greg Speegle: "Formal Aspects of Concurrency Control in Long-Duration Transactions Systems Using the NT/PV Model", *ACM TODS 19,* No. 3 (septiembre, 1994).

Como es mencionado en otros lugares (vea, por ejemplo, las referencias [14.3], [14.9], [14.15] y [14.16]), a menudo la seriabilidad exige imponer una condición a determinados tipos de sistemas para procesamiento de transacciones, en especial en las áreas de aplicación más recientes que involucran la interacción humana y por lo tanto, las transacciones de larga duración. Este artículo presenta un nuevo modelo de transacción llamado NT/PV ("transacciones anidadas con predicados y vistas") que aborda tales preocupaciones. Entre otras cosas muestra que el modelo estándar de transacciones con seriabilidad es un caso especial; define "clases de corrección nuevas y más útiles" y dice que el nuevo modelo proporciona "un marco de trabajo adecuado para la solución de problemas de transacciones de larga duración".

15.14 H. T. Kung y John T. Robinson: "On Optimistic Methods for Concurrency Control", *ACM TODS 6,* No. 2 (junio, 1981).

Podemos describir los esquemas de bloqueo como *pesimistas,* ya que dan por hecho el peor de los casos, en donde cada una de las partes de los datos accedidos por una transacción puede ser requerida por alguna otra concurrente y es mejor que estén bloqueadas. Por el contrario, los esquemas **optimistas,** también conocidos como esquemas de *certificación* o *validación,* hacen la suposición opuesta en donde pareciera que los conflictos son bastante raros en la práctica. Por lo tanto, operan permitiendo que las transacciones se ejecuten completamente libres hasta su terminación y luego hacen una verificación, al momento del COMMIT, para ver si ocurrió algún conflicto. De ser así, la transacción afectada simplemente vuelve a iniciar desde el principio. No se escribe ninguna actualización en la base de datos antes de la terminación satisfactoria del procesamiento de la confirmación, por lo que estos reinicios no requieren que se deshaga ninguna actualización.

Un artículo posterior [15.6] mostró que bajo determinadas suposiciones razonables, métodos optimistas gozan de ciertas ventajas inherentes sobre los métodos de bloqueo tradicionales, en términos del nivel de concurrencia esperada (es decir, la cantidad de transacciones simultáneas) que pueden soportar. Esto sugiere que los métodos optimistas pueden llegar a ser la técnica a elegir en sistemas con múltiples procesadores paralelos. (Por el contrario, la referencia [14.12] indica que los métodos optimistas son de hecho *peores* que el bloqueo en situaciones "de punto caliente", donde un punto caliente es un elemento de datos que es actualizado muy frecuentemente por muchas transacciones distintas. Vea el comentario de la referencia [15.15] para conocer una técnica que funciona bien para los puntos calientes.)

15.15 Patrick E. O'Neil: "The Escrow Transactional Method", *ACM TODS 11,* No. 4 (diciembre, 1986).

Considere el siguiente ejemplo. Supongamos que la base de datos contiene un elemento de datos *TC* que representa el "efectivo total disponible" y suponga que casi todas las transacciones del sistema actualizan a *TC* disminuyéndolo en alguna cantidad (que corresponde, digamos, a un retiro de efectivo). Entonces *TC* es un ejemplo de un "punto caliente", es decir un elemento de la base de datos al que tiene acceso un porcentaje importante de las transacciones que se ejecutan en el sistema. Bajo el bloqueo tradicional, un punto caliente puede convertirse rápidamente en un cuello de botella (para mezclar horriblemente las metáforas). Pero utilizar el bloqueo tradicional sobre un elemento de datos como *TC* es realmente una exageración. Si *TC* tiene inicialmente un valor de diez millones de dólares y cada transacción individual lo disminuye (en promedio) en solamente diez dólares, podríamos ejecutar 1,000,000 de tales transacciones y *además aplicar 1,000,000 de decrementos correspondientes en cualquier orden,* antes de meternos en problemas. Por lo tanto, en realidad no hay necesidad de aplicar ningún bloqueo tradicional a *TC;* todo lo que necesitamos es asegurarnos que el valor actual sea lo suficientemente grande para permitir el decremento requerido y luego hacer la actualización. (Por supuesto, si la transacción falla posteriormente, la cantidad del decremento debe volverse a añadir.)

El método de **custodia** se aplica a situaciones como la que acabo de describir; es decir, situaciones en las que las actualizaciones tienen una forma determinada, en lugar de ser completamente arbitrarias. El sistema debe proporcionar un nuevo tipo de instrucción de actualización especial (por ejemplo "disminuir en *x* si y sólo si el valor actual es mayor que y"). Luego puede realizar la actualización colocando la cantidad *x* "en custodia" y quitándola de la misma al final de la transacción (y confirmando el cambio si el final de la transacción es COMMIT, o sumando la cantidad al total original si el final de la transacción es ROLLBACK).

El artículo describe varios casos en los que se puede usar el método de custodia. Un ejemplo de un producto comercial que soporta la técnica es la versión Fast Path de IMS, de IBM. Hacemos notar que la técnica puede ser vista como un caso especial de control de concurrencia optimista [15.14]. (Sin embargo, observe que el aspecto de "caso especial" —es decir, la provisión de las instrucciones de actualización especial— es crucial.)

15.16 Christos Papadimitriou: *The Theory of Database Concurrency Control.* Rockville, Md.; Compu ter Science Press (1986).

Es un libro de texto que pone énfasis en la teoría formal.

15.17 Kenneth Salem, Hector Garcia-Molina y Jeannie Shands: "Altruistic Locking", *ACM TODS 19,* No. 1 (marzo, 1994).

Propone una extensión al bloqueo de dos fases, según la cual una transacción *TI* que ha terminado de usar alguna parte de los datos que bloqueó, y que no puede desbloquear (debido al protocolo de bloqueo de dos fases), puede sin embargo "donar" los datos al sistema, permitiendo que alguna otra transacción *T2* adquiera un bloqueo sobre ellos. Entonces, se dice que *T2* está "detrás" de 77. Existen protocolos definidos para impedir, por ejemplo, que una transacción vea

cualquier actualización hecha por transacciones que están detrás de ella. Queda mostrado que el bloqueo altruista (el término se deriva del hecho de que la "donación" de datos beneficia a otras transacciones y no a la transacción donante) proporciona más concurrencia que el bloqueo de dos fases convencional, en especial cuando alguna de las transacciones es de larga duración.

RESPUESTAS A EJERCICIOS SELECCIONADOS

15.3

a. Existen seis resultados correctos posibles, que corresponden a los seis planes seriales posibles:

Irucialmente	*A* = 0
T1-T2-T3	*A* - 1
T1-T3-T2	*A* = 2
T2-T1-T3	A = 1
T2-T3-T1	*A* = 2
T3-T1-12	*A* = *A*
T3-T2-T1	*A* = 3

Por supuesto, los seis resultados posibles no son todos distintos. De hecho así sucede en este ejemplo particular, en que todos los resultados correctos posibles son independientes del estado inicial de la base de datos, debido a la naturaleza de la transacción 73.

b. Existen 90 planes distintos posibles. Podemos representar las posibilidades de la manera siguiente. *(Ri, Rj, Rk*representan las tres operaciones de RECUPERACIÓN *Rl,R2y R3,* no necesariamente en ese orden; en forma similar, *Up, Uq, Ur* representan las tres operaciones de ACTUALIZACIÓN *Ul,U2y U3,* tampoco necesariamente en ese orden.)

Ri-Rj-Rk-Up-Uq-Ur :	3 * 2 * 1	= 36	posibilidade s
Ri-Rj-Up-Rk-Uq-Ur :	3 * 2 * 1	= 24	posibilidade
Ri-Rj-Up-Uq-Rk-Ur :	3 * 2 * 1	= 12	posibilidade
Ri~Up-Rj-Rk-Uq-Ur :	3 * 2 * 1	= 12	posibilidade
Ri-Up-Rj-Uq-Rk-Ur :	3 * 2 * 1	= 6	posibilidade s
	TOTAL	= 90	combinacione s

c. Sí. Por ejemplo, el plan *R1-R2-R3-U3-U2-U1* produce el mismo resultado (uno) que dos de los seis planes seriales posibles *{Ejercicio:* Revise este enunciado) y resulta ser "correcto" para el valor inicial dado de cero. Pero debe quedar claro que este "valor correcto" es mera casualidad, y es resultado del simple hecho de que el valor inicial es cero y no algún otro. Como ejemplo contrario, considere lo que sucedería si el valor inicial de *A* fuera diez en lugar de cero. ¿El plan *R1-R2-R3-U3-U2-U1* que mostré anteriormente produciría todavía alguno de los resultados genuinamente correctos? (¿Cuáles *son* los resultados genuinamente correctos en este caso?) De no ser así, ese plan no es seriable.

d. Sí. Por ejemplo, el plan *R1-R3-U1-U3-R2-U2* es seriable (es equivalente al plan serial 77-73-*T2),* pero no puede producirse si 77, *T2 y T3* obedecen el protocolo de bloqueo de dos fases. Porque bajo ese protocolo la operación *R3* adquirirá un bloqueo S sobre A en nombre de la transacción 73; la operación *Ul* de la transacción *TI* no podrá continuar sino hasta que ese blo queo sea liberado, y esto no sucederá sino hasta que la transacción *T3* termine (de hecho, las transacciones *T3* y *TI* quedarán bloqueadas mortalmente al llegar a la operación *U3).*

Este ejercicio ilustra claramente el siguiente punto importante. Dado un conjunto de transacciones y un estado inicial de la base de datos, (a) dejemos que TODO sea el conjunto de todos los

planes posibles que involucran aquellas transacciones; (b) que "CORRECTO" sea el conjunto de todos los planes que garantizan un estado final correcto o al menos que esto suceda desde el estado inicial dado; (c) que SERIABLE sea el conjunto de todos los planes seriables, y (d) que PRODUCIBLE sea el conjunto de todos los planes que pueden producirse bajo el protocolo de bloqueo de dos fases. Entonces, en general:

```
PRODUCIBLE < SERIABLE < "CORRECTO" < TODO
```

(donde "<" significa que "es un subconjunto de").

15.4 En el tiempo *tn,* ¡ninguna transacción está haciendo trabajo útil! Hay un bloqueo mortal que involucra a las transacciones *T2, T3, T9* y *T8;* además *T4* está esperando a T9, *T12* está esperando a *T4, y TÍO y Til* están esperando a 772. Podemos representar la situación por medio de un grafo (el **grafo de espera)** en el cual los nodos representan transacciones y la flecha desde el nodo *Ti* hacia el nodo *Tj* indica que *Ti* está esperando a *Tj* (vea la figura 15.14). Las flechas están etiquetadas con el nombre del elemento de la base de datos y el nivel de bloqueo que están esperando.

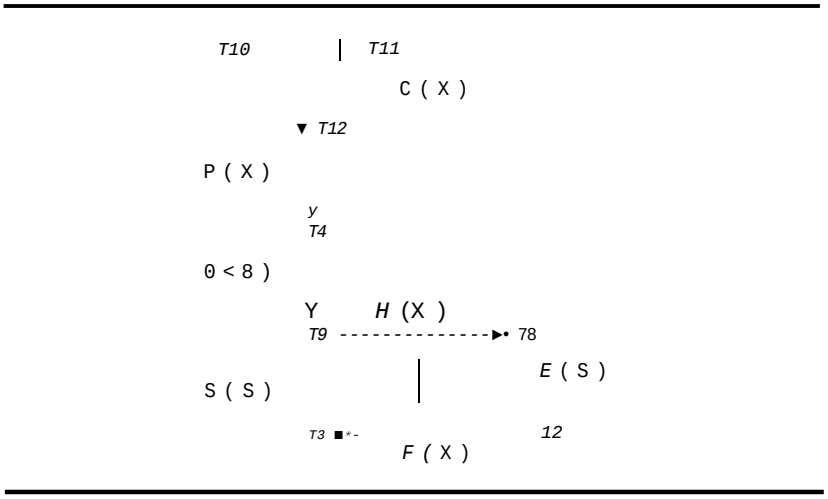


Figura 15.14 El grafo de espera para el ejercicio 15.4.

15.5 El nivel de aislamiento CS tiene el mismo efecto que el nivel de aislamiento RR en los problemas de las figuras 15.1 a 15.3. (Sin embargo, observe que este enunciado *no* se aplica a CS tal como está implementado en DB2, gracias a que en DB2 se utilizan los bloqueos U en lugar de los bloqueos S [4.20].) En lo que se refiere al problema del análisis inconsistente (figura 15.4): el nivel de aislamiento CS no resuelve este problema, ya que la transacción A debe ejecutarse bajo RR para conservar sus bloqueos hasta el final de la transacción; de no ser así, producirá la respuesta equivocada. (En forma alterna, queda claro que *A* podría bloquear toda la varrel de cuentas por medio de alguna petición de bloqueo explícita, en caso de que el sistema soportara esta operación. Esta solución podría funcionar bajo los niveles de aislamiento CS y RR.)

15.6 Vea la sección 15.8. Observe en particular que las definiciones *formales* están dadas por la matriz de compatibilidad de tipos de bloqueo (figura 15.11).

15.9 Los tres problemas de concurrencia identificados en la sección 15.2 fueron *actualización perdida, dependencia no confirmada* y *análisis inconsistente.* De estos tres:

- *Actualizaciones perdidas:* Es necesario que la implementación de SQL garantice (bajo cualquier circunstancia) que nunca sucedan las actualizaciones perdidas.
- *Dependencia no confirmada:* Éste es simplemente otro nombre para la lectura sucia.
- *Análisis inconsistente:* Este término abarca a las lecturas no repetibles y a los fantasmas.

15.10 La siguiente descripción está tomada de la referencia [20.15]. En primer lugar, el sistema debe mantener:

1. Para cada objeto de datos, una pila de versiones confirmadas (cada entrada de la pila debe dar un valor para el objeto y el ID de la transacción que estableció ese valor; es decir, cada entrada de la pila consta esencialmente de un apuntador hacia la entrada relevante en la bitácora). La pila se encuentra en secuencia cronológica inversa, con la entrada más reciente hasta arriba.
2. Una lista de los IDs de transacción para todas las transacciones confirmadas (la *lista de confirmación).*

Cuando una transacción inicia su ejecución, el sistema le proporciona una copia privada de la lista de confirmación. Las operaciones de lectura sobre un objeto están dirigidas hacia la versión más reciente del objeto producido por una transacción que está en esa lista privada. Las operaciones de actualización, por el contrario, están dirigidas hacia el objeto de datos actual (y ésta es la razón por la que las pruebas de conflictos de actualización/actualización siguen siendo necesarias). Cuando la transacción es confirmada, el sistema actualiza adecuadamente la lista de confirmación y las pilas de las versiones de los objetos de datos.

PARTE V

TEMAS ADICIONALES

En la parte II de este libro afirmamos que el modelo relacional es el fundamento de la tecnología moderna de bases de datos; y así es. Sin embargo, es *solamente* el fundamento: existe mucho más en la tecnología de bases de datos que el simple modelo relacional, y los estudiantes y profesionales de las bases de datos necesitan estar familiarizados con muchos conceptos y características adicionales para estar completamente "al tanto de las bases de datos" (como debe haber quedado claro, por lo que he explicado en las partes III y IV). Ahora nos enfocaremos en otros temas importantes. En secuencia, los temas a tratar son los siguientes:

- Seguridad (capítulo 16)
- Optimization (capítulo 17)
- Información faltante (capítulo 18)
- Herencia de tipo (capítulo 19)
- Bases de datos distribuidas (capítulo 20)
- Apoyo para la toma de decisiones (capítulo 21)
- Bases de datos temporales (capítulo 22)
- Bases de datos basadas en la lógica (capítulo 23)

De hecho, la secuencia anterior es un poco arbitraria, sin embargo he escrito los capítulos con la suposición de que serán leídos (tal vez en forma selectiva) en el orden en que aparecen.

CAPITULO 16

Seguridad

16.1 INTRODUCCIÓN

La cuestión de la seguridad de los datos se asocia frecuentemente con la de la integridad de los mismos (al menos en contextos informales), pero ambos conceptos son bastante diferentes. La *seguridad* se refiere a la protección de los datos contra su revelación, su alteración o su destrucción no autorizadas, mientras que la *integridad* se refiere a la precisión o validez de esos datos. Para ponerlo un poco más claro:

- **Seguridad** significa proteger los datos ante usuarios no autorizados.
- **Integridad** significa protegerlos de usuarios autorizados.

O dicho más a la ligera: la seguridad significa garantizar que los usuarios tengan *permiso* de hacer las cosas que están tratando de hacer y la integridad involucra asegurar que las cosas que están tratando de hacer sean *correctas.*

Por supuesto, también existen algunas similitudes: en ambos casos el sistema necesita estar al tanto de ciertas *restricciones* que los usuarios no deben violar y en ambos casos, estas restricciones deben ser especificadas (generalmente por el DBA) en un lenguaje adecuado y deben ser mantenidas en el catálogo del sistema; también, en ambos casos el DBMS debe vigilar las operaciones del usuario para asegurarse que cumplan estas restricciones. En este capítulo examinamos específicamente la seguridad (por supuesto, ya hemos tratado ampliamente el tema de la integridad en el capítulo 8 y en otras partes). *Nota:* La razón principal por la que separamos las explicaciones de ambos temas, es porque consideramos a la integridad como un asunto absolutamente fundamental y la seguridad como un asunto más secundario.

Consideraciones generales

Existen muchos aspectos sobre el problema de la seguridad, entre ellos se encuentran los siguientes:

- Aspectos legales, sociales y éticos (por ejemplo, la persona que hace la petición ¿tiene dere cho legal para conocer la información solicitada como, digamos, el crédito de un cliente?);
- Controles físicos (por ejemplo, ¿el lugar en donde se encuentra la computadora o terminal está bajo llave o con alguna otra protección?);
- Cuestiones de política (por ejemplo, ¿cómo decide la empresa propietaria del sistema a quién y a qué se le permite tener acceso?);
- Problemas operacionales (por ejemplo, si se utiliza un esquema de contraseñas, ¿cómo se les mantiene en secreto? ¿con cuánta frecuencia son cambiadas?);
- Controles de hardware (por ejemplo, ¿la unidad de procesamiento proporciona alguna ca racterística de seguridad, como claves de protección de almacenamiento o un modo de operación protegido?);

- Soporte del sistema operativo (por ejemplo, ¿el sistema operativo subyacente borra el con tenido de la memoria principal y los archivos de disco cuando ha terminado de utilizarlos?);

y por último

- Los asuntos que conciernen únicamente al sistema de base de datos (por ejemplo, ¿tiene el sistema de base de datos un concepto de propiedad de los datos?).

Por razones obvias, en este capítulo limitaremos nuestra atención únicamente a los asuntos de esta última categoría (en su mayoría).

Actualmente los DBMSs modernos soportan generalmente uno o ambos enfoques con respecto a la seguridad de los datos. Estos enfoques son conocidos como control *discrecional* y control *obligatorio,* respectivamente. En ambos casos, la unidad de datos u "objeto de datos" que necesite ser protegido puede comprender desde toda la base de datos (por un lado) hasta un componente específico dentro de una tupia específica (por otro). La manera en que difieren los dos enfoques está indicada brevemente por el siguiente esquema:

- En el caso del control **discrecional,** un usuario específico tendrá generalmente diferentes derechos de acceso (también conocidos como **privilegios)** sobre diferentes objetos; además, existen muy pocas limitaciones —es decir, limitaciones inherentes— sobre qué usuarios pueden tener qué derechos sobre qué objetos (por ejemplo, el usuario *Ul* podría estar au torizado para ver a *A* y no a *B;* y en cambio, el usuario *U2* podría estar autorizado para ver a *B* y no a A). Por lo tanto, los esquemas discrecionales son muy flexibles.
- Por el contrario, en el caso del control **obligatorio,** cada objeto de datos está etiquetado con un nivel de clasificación determinado y a cada usuario se le da un nivel de **acreditación.** Un objeto de datos específico sólo puede ser accedido por los usuarios que tengan el nivel de acreditación adecuado. Por lo tanto, los esquemas obligatorios tienden a ser jerárquicos por naturaleza y (por lo tanto) comparativamente rígidos. (Si el usuario *Ul* puede ver a A pero no a *B,* entonces la clasificación de *B* debe ser más alta que la de A y por lo tanto, ningún usuario *U2* podrá ver a *B* y no a A.)

En la sección 16.2 explicamos los esquemas discrecionales y en la sección 16.3 los obligatorios. Independientemente de que estemos tratando con un esquema discrecional o con uno obligatorio, todas las decisiones sobre a qué usuarios se les permite realizar qué operaciones sobre qué objetos, son decisiones políticas y no técnicas. Como tales, están claramente fuera de la jurisdicción del propio DBMS, ya que al DBMS sólo le queda hacer cumplir esas decisiones una vez que se han tomado. De aquí que:

- Los resultados de esas decisiones políticas (a) deben ser del conocimiento del sistema (esto se logra por medio de instrucciones en un lenguaje de definición adecuado) y (b) deben ser recordadas por el sistema (esto se logra guardándolas en el catálogo).
- Debe existir algún medio para revisar una cierta petición de acceso contra las restricciones de seguridad aplicables que están en el catálogo. (En general, "petición de acceso" significa la combinación de la *operación solicitada* más el *objeto solicitado* más el *usuario que lo solicita.)* Esa revisión es realizada por el **subsistema de seguridad** del DBMS, al que tam bién se conoce como subsistema de **autorización.**
- Para decidir qué restricciones de seguridad son aplicables a una cierta petición de acceso, el sistema debe ser capaz de reconocer el *origen* de esa petición; es decir, debe ser capaz de reconocer al *usuario solicitante.* Por esta razón, cuando los usuarios se registran en el sistema

se les pide generalmente que proporcionen no solamente su ID de usuario (para que digan quiénes son), sino también una **contraseña** (para probar que son quienes dicen ser). Se supone que la contraseña es conocida únicamente por el sistema y por los usuarios legítimos del ID al que se refiere.*

Dicho sea de paso —con relación a este último punto— tenga presente que cualquier cantidad de usuarios distintos pueden compartir el mismo ID. De esta forma, el sistema puede soportar **grupos de usuarios** y por lo tanto, permitir (digamos) que todos los usuarios del departamento de contabilidad compartan los mismos privilegios sobre los mismos objetos. Entonces, las operaciones de incorporación o eliminación de usuarios individuales de un grupo determinado, pueden ser realizadas en forma independiente de la operación de especificar qué privilegios sobre qué objetos se aplican para ese grupo. Sin embargo, observe que el lugar obvio para conservar un registro de qué usuarios están en cada grupo, es una vez más el catálogo (o tal vez la propia base de datos). Al respecto, le pedimos que vea la referencia [16.9], la cual describe un sistema en el que los grupos de usuarios pueden estar *anidados.* Para citar: "La habilidad para clasificar usuarios en una jerarquía de grupos proporciona una poderosa herramienta para la administración de sistemas grandes con miles de usuarios y objetos."

16.2 CONTROL DE ACCESO DISCRECIONAL

Para repetir lo que hemos dicho en la sección anterior, la mayoría de los DBMSs soportan el control discrecional, el control obligatorio o ambos. De hecho, sería más apropiado decir que la mayoría de los sistemas soportan el control discrecional y algunos sistemas soportan también el control obligatorio; y puesto que en la práctica es más común encontrar el control discrecional, lo abordaremos primero.

Como ya observamos, necesitamos un lenguaje que soporte la definición de las restricciones de seguridad (discrecionales). Sin embargo, por razones bastante obvias, es más fácil decir lo que *está permitido* en vez de lo que *no* lo está; por lo tanto, los lenguajes por lo general soportan la definición no de las restricciones de seguridad como tales, sino de las **autoridades,** las cuales son en efecto lo opuesto a las restricciones de seguridad (si algo está autorizado, no está restringido). Por lo tanto, comenzaremos describiendo brevemente un lenguaje para la definición de autoridades^ Éste es un primer ejemplo sencillo:

```
AUTHORITY AV3
   GRANT RETRIEVE ( V#, PROVEEDOR, CIUDAD ), DELETE
   ON    V
   TO    Jim, Fred, Mary ;
```

---

* Al hecho de verificar que los usuarios sean quienes dicen ser, se le llama autentificación. Observamos de paso que las técnicas de autentificación que están disponibles actualmente son mucho más sofisticadas que la simple revisión de contraseñas, ya que involucran una diversidad de dispositivos biométricos (lectores de huellas digitales, escáneres de retina, generadores de imágenes para geometría de la mano, verificadores de voz, reconocedores de firmas, etcétera). Todos estos dispositivos pueden ser usados de manera efectiva para verificar las "características personales que nadie puede robar" [16.3].

[tr]Tutorial D, tal como fue definido originalmente en [3.3], no incluye deliberadamente ninguna propiedad para definición de autoridad, pero el lenguaje hipotético presentado en esta sección puede ser considerado como si fuera el "espíritu" del Tutorial D.

Este ejemplo pretende ilustrar el punto de que (por lo general) las autoridades tienen *cuatro componentes,* que son los siguientes:

1. Un **nombre** (AV3 en el ejemplo). La autoridad será registrada en el catálogo bajo este nombre.
2. Uno o más **privilegios** (RETRIEVE —sólo sobre ciertos atributos— y DELETE) especifi cados por medio de la cláusula GRANT.
3. La **varrel** a la que se aplica la autoridad (la varrel V), especificada por medio de la cláu sula **ON.**
4. Uno o más **"usuarios"** (para ser más precisos, *IDs de usuario)* a quienes se van a otorgar los privilegios especificados sobre la varrel especificada, dados por medio de la cláusula TO.

Entonces, la sintaxis general es:

```
AUTHORITY <nombre de autoridad>
    GRANT  <lista de privilegios separados con comas>
    ON     <nombre de varrel>
    TO     <lista de IDs de usuario separados con comas>  ;
```

Explicación: <nombre de autoridad>, <nombre de varrel> y *<lista de IDs de usuario separados con comas>* son muy claras (con excepción de que consideramos ALL, que significa todos los usuarios conocidos, como un "ID de usuario" válido en este contexto). Cada *<privilegio>* es alguno de los siguientes:

```
RETRIEVE [   ( <lista de nombres de atributo separados con comas>  )]
INSERT     [   (  <lista de nombres de atributo separados con comas>  )]
UPDATE     [   (  <lista de nombres de atributo separados con comas>  )]
DELETE ALL
```

RETRIEVE (sin calificativos), INSERT (sin calificativos), UPDATE (sin calificativos) y DELETE son muy claras.* Si con RETRIEVE se especifica una lista de nombres de atributos separados con comas, entonces el privilegio se aplica solamente a los atributos especificados; INSERT y UPDATE con una lista de nombres de atributos separados con comas se define en forma similar. La especificación ALL es una abreviatura para todos los privilegios: RETRIEVE (todos los atributos), INSERT (todos los atributos), UPDATE (todos los atributos) y DELETE. *Nota:* Por razones de simplicidad, ignoramos el hecho de si es necesario algún privilegio especial para realizar las operaciones generales de asignación relacional. También limitamos deliberadamente nuestra atención a las operaciones de *manipulación de datos;* aunque por supuesto, en la práctica existen muchas otras operaciones que también querremos tener sujetas a verificación de autorización (tales como las operaciones de definición y eliminación de varrels y las operaciones para definir y eliminar a las autoridades mismas). Por razones de espacio, aquí omitimos las consideraciones detalladas sobre tales operaciones.

¿Qué sucedería si algún usuario intentara realizar alguna operación sobre algún objeto y no tiene autorización para ello? Obviamente, la opción más simple es rechazar el intento (y por supuesto, proporcionar la información de diagnóstico adecuada); seguramente dicha respuesta

*Bueno, tal vez no tanto, ya que el privilegio RETRIEVE también es necesario simplemente para *mencionar* al objeto relevante (por ejemplo, en una definición de vista o en una restricción de integridad), así como para recuperarlo como tal.

será comúnmente la más requerida en la práctica y por lo tanto, podemos hacer que sea la predeterminada. Sin embargo, en situaciones más delicadas, tal vez sea más adecuado aplicar otras medidas (por ejemplo, terminar el programa o bloquear el tecleado del usuario). Probablemente también sea necesario registrar tales intentos en una bitácora especial *{vigilancia de amenazas)* para permitir un análisis posterior de los intentos de violación a la seguridad y también para que sirva —por sí misma— como una medida disuasiva contra la infiltración ilegal (vea el comentario sobre los *registros de auditoría* al final de esta sección). Por supuesto, también necesitamos una forma para eliminar autoridades:

```
DROP AUTHORITY <nombre de autoridad* ;
```

Por ejemplo:

```
DROP AUTHORITY AV3 ;
```

Por razones de simplicidad, suponemos que la eliminación de una varrel específica ocasionará la eliminación automática de todas las autoridades que se apliquen sobre esa varrel. Estos son algunos ejemplos adicionales de autoridades, y la mayoría de ellos son muy claros.

1.
```
AUTHORITY EX1
   GRANT RETRIEVE ( P#, PARTE, PESO )
   ON    P
   TO    Jacques, Anne, Charley ;
```

   Los usuarios Jacques, Anne y Charley pueden ver un "subconjunto vertical" de la varrel base P. El ejemplo es entonces un ejemplo de una autoridad **independiente del valor.**

2.
```
AUTHORITY EX2
   GRANT RETRIEVE, UPDATE ( PROVEEDOR, STATUS ), DELETE
   ON    VL
   TO    Dan, Misha ;
```

   Aquí, la varrel VL es una vista ("proveedores de Londres", vea la figura 9.4 del capítulo 9). Los usuarios Dan y Misha pueden ver entonces un "subconjunto horizontal" de la varrel base V. Éste es un ejemplo de una autoridad **dependiente del valor.** Observe también que aunque los usuarios Dan y Misha pueden aplicar DELETE sobre determinadas tupias de proveedor (por medio de la vista VL), no pueden aplicarles INSERT ni pueden aplicar UPDATE sobre atributos de V# o CIUDAD.

3.
```
VAR VVPPR VIEW
   ( V JOIN VP JOIN ( P WHERE CIUDAD = 'Roma' ) { P# } )
                                          { ALL BUT P#, CANT } ;


AUTHORITY EX3
   GRANT RETRIEVE
   ON    VVPPR
   TO    Giovanni ;
```

   Éste es otro ejemplo dependiente del valor: El usuario Giovanni puede recuperar información del proveedor, pero sólo para los proveedores que proporcionan alguna parte que está almacenada en Roma.

```
4. VAR VVC VIEW
       SUMMARIZE VP PER V { V# } ADD SUM ( CANT ) AS VC ;

   AUTHORITY EX4
      GRANT RETRIEVE
      ON VVC TO
      Fidel ;
```

El usuario Fidel puede ver el número de envíos totales por proveedor, pero no el número de embarques individuales. Por lo tanto, el usuario Fidel ve un **resumen estadístico** de los datos base subyacentes.

```
5.
AUTHORITY EX5
  GRANT  RETRIEVE,   UPDATE  STATUS )
                                                 'Thu'    'Fri'
  ON     V
  WHEN   DAY () IN     (     'Tue', 'Wed
         AND NOW ()  > TIME  '09:00:00'
         AND NOW ()  < TIME  '17:00:00'
  TO     Compras ;
```

Aquí extendemos la sintaxis de AUTHORITY para incluir una cláusula WHEN que especifica determinados "controles de contexto" y también estamos suponiendo que el sistema proporciona dos operadores niládicos —es decir, operadores que no toman operandos— llamados DAY() y NOW(), que tienen interpretaciones obvias. La autoridad EX5 garantiza que los valores de status de los proveedores pueden ser cambiados por el usuario "Compras" (que identifica presuntamente a cualquier persona del departamento de compras) sólo en un día laboral y sólo durante las horas hábiles. Éste es un ejemplo de lo que en ocasiones se conoce como autoridad **dependiente del contexto,** ya que una determinada petición de acceso será permitida o no dependiendo del contexto (que en este caso es la combinación de día de la semana y hora del día en el que es emitido).

Otros ejemplos de operadores integrados que probablemente deba soportar de alguna forma el sistema y que podrían ser útiles para las autoridades dependientes del contexto, incluyen:

TODAY o — valor = fecha actual
USER() — valor = el ID del usuario actual
TERMÍNALO — valor = el ID de la terminal donde se origina la petición actual

Para estos momentos, tal vez ya haya notado que —desde un punto de vista conceptual— todas las autoridades se ven afectadas por "OR". En otras palabras, una cierta petición de acceso (que significa, para repetir, la combinación de la operación solicitada, más el objeto solicitado, más el usuario que lo solicita) es aceptable si y sólo si al menos una autoridad lo permite. Sin embargo, observe que (por ejemplo) si (a) una autoridad permite que el usuario Nancy recupere los colores de las partes y (b) otra permite que recupere el peso de las partes, no podemos concluir que el usuario pueda recuperar los colores y los pesos de las partes al mismo tiempo (se necesita una autoridad independiente para la combinación).

Por último, ha quedado implícito (aunque no está de más decirlo) que los usuarios sólo pueden hacer las cosas que tienen permitidas explícitamente por las autoridades definidas. Cualquier cosa que no esté autorizada explícitamente, ¡está prohibida implícitamente!

Modificación de la petición

Para ilustrar algunas de las ideas que presentamos anteriormente, describiremos brevemente los aspectos de seguridad del prototipo del University Ingres y su lenguaje de consulta QUEL, ya que adopta un enfoque interesante ante el problema. En esencia, cualquier petición QUEL se modifica automáticamente antes de su ejecución, de tal forma que es imposible que viole alguna restricción de seguridad especificada. Por ejemplo, supongamos que al usuario *U* se le permite recuperar partes guardadas solamente en Londres:

```
DEFINE PERMIT RETRIEVE ON P TO U
       WHERE P.CIUDAD = "Londres"
```

(vea más adelante los detalles de la operación DEFINE PERMIT). Supongamos ahora que el usuario *U* emite la petición QUEL:

```
RETRIEVE ( P.P#, P.PESO )
WHERE    P.COLOR = "Rojo"
```

Si utilizamos el "permiso" para la combinación de la varrel P y el usuario *U* tal como están guardados en el catálogo, el sistema modifica automáticamente la petición para que sea vista de esta forma:

```
RETRIEVE ( P.P#, P.PESO ) WHERE
P.COLOR = "Rojo" AND
P.CIUDAD = "Londres"
```

Y por supuesto, esta petición modificada no puede violar la restricción de seguridad. Observe de paso que el proceso de modificación es "silencioso"; de hecho, al usuario *U* no se le informa cuando el sistema ejecuta una instrucción ligeramente diferente a la petición original, ya que este hecho en sí puede ser sensible (al usuario *U* tal vez ni siquiera se le permita saber que hay partes que no son de Londres).

El proceso de *modificación de petición* que acabamos de ilustrar es de hecho idéntico a la técnica que se usa para implementar las vistas [8.20] y también —específicamente en el caso del prototipo Ingres— las restricciones de integridad [9.15]. Por lo tanto, una ventaja del esquema es que es muy fácil de implementar, ya que gran parte del código necesario ya existe en el sistema. Otra es que es comparativamente eficiente, ya que el reforzamiento principal de la seguridad sucede en tiempo de compilación y no en tiempo de ejecución (o al menos en parte). Otra ventaja adicional es que no se presentan algunas de las dificultades que pueden ocurrir con el enfoque de SQL cuando un usuario determinado necesita privilegios diferentes sobre partes diferentes de la misma varrel (vea la sección 16.6).

Una desventaja es que no todas las restricciones de seguridad se pueden manejar de manera tan sencilla. A manera de ejemplo, supongamos que al usuario *U* no se le permite ningún acceso a la varrel P. Por lo tanto, ninguna forma simple "modificada" del RETRIEVE mostrado anteriormente, puede mantener la ilusión de que no existe la varrel P. En su lugar, es necesario producir un mensaje de error explícito del tipo "usted no tiene permitido el acceso a esta varrel". (O tal vez el sistema simplemente podría *mentir* y decir "no existe esta varrel".)

Ésta es entonces la sintaxis de DEFINE PERMIT:

```
DEFINE PERMIT <lista de nombres de operación separados con comas>
       ON     <nombre de varrel> [ ( <lista de nombres de atributo separados con
                comas> ) ]
       TO     <ID de usuario>
     [ AT     <lista de nombres de terminal separados con comas> ]
     [ FROM   <tiempo> TO <tiempo> ]
     [ ON     <dia> TO <dia> ]
     [ WHERE  <expresión lógica> ]
```

Esta instrucción es —en forma conceptual— bastante similar a nuestra instrucción AUTHORITY; con la excepción de que soporta una cláusula WHERE. He aquí un ejemplo:

```
    DEFINE PERMIT APPEND, RETRIEVE, REPLACE
ON    V ( V#, CIUDAD
TO    Joe
AT    TTA4
FROM  9:00 TO 17:00
ON    Sat TO Sun
WHERE V.STATUS < 50
AND   V.V# = VP.V#
AND   VP.P# ■ P.P#
AND   P.COLOR = "Rojo
```

Nota: APPEND y REPLACE son los equivalentes de QUEL para nuestros INSERT y UPDATE, respectivamente.

Registros de auditoría

Es importante no dar por hecho que el sistema de seguridad es perfecto. Un posible infiltrador que tenga la determinación suficiente, encontrará generalmente una forma de pasar por encima de los sistemas de control (en especial si la recompensa es alta). Por lo tanto, en las situaciones donde los datos son muy delicados, o cuando el procesamiento que se realiza sobre éstos es lo suficientemente crítico, un **registro de auditoría** llega a ser una necesidad. Por ejemplo, si las discrepancias de los datos conducen a la sospecha de que la base de datos ha sido alterada, el registro de auditoría puede ser usado para examinar lo que ha estado sucediendo y para verificar que todo está bajo control (o para ayudar a señalar dónde hubo un error).

En esencia, un registro de auditoría es un archivo o base de datos especial en el que el sistema lleva automáticamente la cuenta de todas las operaciones realizadas por los usuarios sobre los datos normales. En algunos sistemas el registro de auditoria puede estar integrado físicamente con la bitácora de recuperación (vea el capítulo 14), mientras que en otros, los dos pueden ser distintos pero los usuarios deben —de cualquier forma— tener la posibilidad de consultar el registro de auditoría usando su lenguaje de consulta normal (por supuesto, siempre y cuando tengan autorización). Un registro de auditoría típico podría contener la siguiente información:

Petición (texto de origen)

Terminal desde la que se llamó a la operación

Usuario que llamó a la operación

Fecha y hora de la operación

Varrels, tupias, atributos afectados

Valores antiguos Valores nuevos

Como mencioné anteriormente en esta sección, el simple hecho de mantener un registro de auditoría puede ser suficiente para disuadir en algunas situaciones a un posible infiltrador.

16.3 CONTROL DE ACCESO OBLIGATORIO

Los controles obligatorios son aplicables a las bases de datos en las que los datos tienen una estructura de clasificación bastante estática y rígida, como puede ser el caso de (por ejemplo) determinados ambientes militares o gubernamentales. Como expliqué brevemente en la sección 16.1, la idea básica es que cada objeto de datos tiene un **nivel de clasificación** (por ejemplo, supersecreto, secreto, confidencial, etcétera) y cada usuario tiene un **nivel de acreditación** (con las mismas posibilidades que tienen los niveles de clasificación). Se supone que los niveles forman un ordenamiento estricto (por ejemplo, supersecreto > secreto > confidencial, etcétera). Por lo tanto, se imponen las siguientes reglas que se deben a Bell y La Padula [16.1]:

1. El usuario ¡ puede recuperar el objeto 7 sólo si el nivel de acreditación de *i* es mayor o igual al nivel de clasificación de *j* (la "propiedad de seguridad simple");
2. El usuario *i* puede actualizar el objeto./ sólo si el nivel de acreditación de í es igual al nivel de clasificación *dej* (la "propiedad de estrella").

La primera regla es suficientemente obvia, pero la segunda requiere una explicación. Observe primero que otra forma de establecer la segunda regla es decir que, por definición, cualquier cosa escrita por el usuario *i* adquiere automáticamente un nivel de clasificación igual al nivel de acreditación de *i.* Dicha regla es necesaria para impedir que un usuario con clasificación, por ejemplo, "secreta" copie datos secretos hacia un archivo que tenga una clasificación menor, minando por lo tanto el propósito del esquema de clasificación. *Nota:* Desde el punto de vista de las operaciones de "escritura" (INSERT) puras, sería suficiente que la segunda regla dijera que el nivel de acreditación de ; debe ser *menor o* igual al nivel de clasificación *dej;* y la regla en ocasiones aparece de esta forma en la literatura. Pero entonces los usuarios, ¡serían capaces de escribir cosas que no podrían leer! (pensándolo bien, algunas personas tienen dificultades para leer lo que ellas mismas escriben... tal vez la regla más débil no esté tan fuera de la realidad después de todo).

Los controles obligatorios comenzaron a recibir mucha atención en el mundo de las bases de datos a principios de los años noventa, ya que fue entonces cuando el Departamento de Defensa de los Estados Unidos comenzó a requerir que cualquier sistema que adquiriera tuviera soporte para tales controles. En consecuencia, los fabricantes de DBMS han estado compitiendo entre ellos para implementarlos. Los controles en cuestión están documentados en dos publicaciones importantes del Departamento de Defensa conocidas informalmente como el **Libro naranja [16.19] y** el **Libro lavanda** [16.20], respectivamente. El Libro naranja define un conjunto de requerimientos de seguridad para cualquier "base de computación confiable" (TCB), mientras que el Libro lavanda define específicamente una "interpretación" de los requerimientos TCB para los sistemas de base de datos.

De hecho, los controles obligatorios definidos en las referencias [16.19 y 16.20] forman parte de un esquema más general de clasificación de seguridad total, que resumimos aquí para

efectos de referencia. En primer lugar, los documentos definen cuatro **clases de seguridad** (D, C, B y A); en términos generales, la clase D es la menos segura, la clase C es más segura que la clase D y así sucesivamente. Decimos que la clase D proporciona protección *mínima,* la clase C protección *discrecional,* la clase B protección *obligatoria* y la clase A protección *verificada.*

- **Protección discrecional:** La clase C está dividida en dos subclases, Cl y C2 (donde Cl es *menos* segura que C2). Cada una de ellas soporta controles discrecionales, lo que significa que el acceso está sujeto a la discreción del *propietario* de los datos (como vimos anterior mente en la sección 16.2). Además:
  1. La clase Cl distingue entre propiedad y acceso; es decir, soporta el concepto de datos compartidos, permitiendo al mismo tiempo que los usuarios también tengan datos priva dos propios.
  2. La clase C2 requiere —adicionalmente— de soportes de contabilización por medio de procedimientos de registro, auditoría y aislamiento de recursos.
- **Protección obligatoria:** La clase B es aquella que maneja los controles obligatorios. Está dividida adicionalmente en subclases Bl, B2 y B3 (en donde Bl es la *menos* segura y B3, la más) de la siguiente manera:
  1. La clase Bl requiere "protección de seguridad etiquetada" (es decir, requiere que cada objeto de datos esté etiquetado con su nivel de clasificación: secreto, confidencial, etcé tera). También requiere, en efecto, una instrucción informal de la política de seguridad.
  2. La clase B2 requiere además una instrucción *formal* de lo mismo. También requiere que los *canales cubiertos* sean identificados y eliminados. Ejemplos de canales cubiertos pudieran ser (a) la posibilidad de inferir la respuesta a una consulta no válida a partir de la respuesta de una válida (vea la sección 16.4) o (b) la posibilidad de deducir informa ción sensible a partir del tiempo que requiere la realización de algún cálculo válido (vea el comentario a la referencia [16.12]).
  3. La clase B3 requiere específicamente el soporte de auditoria y recuperación, así como la designación de un *administrador de seguridad.*
- **Protección verificada:** La clase A (la más segura) requiere una prueba matemática de que (a) el mecanismo de seguridad es consistente y que (b) es adecuado para soportar la política de seguridad especificada (!).

Varios productos DBMS comerciales proporcionan actualmente controles obligatorios a nivel Bl. También proporcionan —por lo general— controles discrecionales a nivel C2. *Terminología:* a los DBMS que soportan controles obligatorios se les llama en ocasiones sistemas **seguros de múltiples niveles** [16.13,16.16,16.21] (vea la subsección que viene a continuación). El término sistema **confiable** también es usado casi con el mismo significado [16.17, 16.19, 16.20].

Seguridad de múltiples niveles

Suponga que queremos aplicar las ideas del control de acceso obligatorio a la varrel V de proveedores. Por razones de claridad y simplicidad, suponga que la unidad de datos sobre la cual queremos controlar el acceso es la tupia individual dentro de esa varrel. Entonces cada tupia necesita estar etiquetada con su nivel de clasificación, tal vez como se muestra en la figura 16.1 (4 = supersecreto, 3 = secreto, 2 = confidencial, etcétera).

V

V#	PROVEEDOR	STATUS	CIUDAD	CLASE
V1	Smith	20	Londres	2
V2	Jones	10	París	3
V3	Blake	30	París	2
V4	Clark	20	Londres	4
V5	Adams	30	Atenas	3

Figura 16.1 La varrel V con niveles de clasificación (ejemplo).

Ahora, supongamos que los usuarios *Ul y U2* tienen niveles de acreditación 3 (secreto) y 2 (confidencial), respectivamente. Entonces, ¡los usuarios *Ul y* í/2 verán a la varrel V de manera diferente! Una petición para recuperar todos los proveedores regresará cuatro tupias (las correspondientes a VI, V2, V3 y V5) si es emitida por *Ul*, pero sólo dos tupias (las de VI y V3) si es emitida por *U2*. Los que es más, *ninguno* de los usuarios verá la tupia para V4.

Una forma de pensar sobre lo anterior es de nuevo en términos de la *modificación de la petición*. Considere la siguiente consulta ("obtener los proveedores de Londres"):

```
V WHERE CIUDAD = 'Londres'
```

El sistema modificará esta petición para que se vea de esta forma:

```
V WHERE CIUDAD • 'Londres' AND CLASE < acreditación del usuario
```

Consideraciones similares se aplican a las operaciones de actualización. Por ejemplo, el usuario *Ul* no está consciente de que exista la tupia para V4. Por lo tanto, para ese usuario el siguiente INSERT le parece razonable:

```
INSERT INTO V RELATION { TUPLE { V#       V# ( 'V4' ),
                                 PROVEEDOR NOMBRE ( 'Baker' ),
                                 STATUS   25,
                                 CIUDAD   'Roma' } } ;
```

El sistema no debe rechazar este INSERT, ya que al hacerlo le diría efectivamente al usuario *Ul* que el proveedor V4 sí existe después de todo. Por lo tanto, lo acepta, pero lo modifica para que sea:

```
INSERT INTO V RELATION { TUPLE { V#       V# ( 'V4' ),
                                 PROVEEDOR NOMBRE ( 'Baker' ),
                                 STATUS   25,
                                 CIUDAD   'Roma',
                                 CLASE    CLASE ( 3 ) } } ;
```

Por lo tanto, observe que la clave primaria para los proveedores no es solamente {V#}, sino la combinación {V#, CLASE). *Nota:* Suponemos, por razones de simplicidad, que sólo existe una clave candidata, la cual, por lo tanto, podemos ver (sin peligro) como la clave *primaria.*

Más terminología: la varrel de proveedores es un ejemplo de una **varrel de múltiples niveles.** El hecho de que el "mismo" dato aparezca diferente ante diferentes usuarios es llamado **polinstanciación.** Después del INSERT que acabamos de explicar, por ejemplo, una petición

para recuperar al proveedor V4 regresa un resultado a un usuario que tenga acreditación super-secreta, otra al usuario *Ul* (con acreditación secreta) y otra más al usuario *U2* (con acreditación confidencial).

UPDATE y DELETE son tratados en forma similar (aquí omitimos los detalles, pero tome en cuenta que muchas de las referencias al final del capítulo tratan estos temas a profundidad). Una pregunta: ¿cree usted que las ideas que se han tratado anteriormente constituyen una violación al *Principio de la información!* Justifique su respuesta.

1 BASES DE DATOS ESTADÍSTICAS

Nota: Gran parte del material de esta sección y la siguiente apareció originalmente —un poco diferente— en la referencia [16.4].

Una *base de datos estadística* (en el contexto que estamos considerando) es aquella que permite consultas que proporcionen información general (por ejemplo sumas, promedios) pero no consultas que proporcionen información individual. Por ejemplo, la consulta "¿cuál es el salario promedio de los programadores?" podría ser permitida y en cambio, la consulta "¿cuál es el salario de la programadora Mary?", no.

El problema con estas bases de datos es que en ocasiones es posible hacer inferencias a partir de consultas válidas para deducir las respuestas a consultas no válidas. Como lo dice la referencia [16.6]: "los resúmenes contienen vestigios de la información original; un fisgón podría (re)construir esta información procesando resúmenes suficientes. A esto se le llama *deducción de información confidencial por inferencia".* Remarcamos que este problema puede llegar a ser cada vez más importante en la medida en que se incremente el uso de los *data warehouse* (vea el capítulo 21).

He aquí un ejemplo detallado. Suponga que la base de datos contiene solamente una varrel, STATS (vea la figura 16.2). Suponga, por simplicidad, que todos los atributos están definidos sobre tipos de datos primitivos (en esencia, números y cadenas). Suponga además que algún

NOMBRE	SEXO	HIJOS	OCUPACIÓN	SALARIO	IMPUESTOS	AUDITORIAS
Alf	M	3	Programador	50K	10K	3
Bea	F	2	Médico	130K	10K	0
Cary	F	a	Programador	56K	18K	1
Dawn	F	2	Constructor	60K	12K	1
Ed	H	2	Empleado	44K	4K	0
Fay	F	1	Artista	30K	0K	0
Guy	H	0	Abogado	190K	0K	0
Hal	M	3	Constructor de casas	44K	2K	0
Ivy	F	4	Programador	64K	10K	1
Joy	F	1	Programador	60K	20K	1

Figura 16.2 La varrel STATS (valores de ejemplo).

usuario *U* está autorizado para realizar (solamente) consultas estadísticas y pretende descubrir el salario de Alf. Por último, supongamos que *U* sabe, por fuentes externas, que Alf es un programador de sexo masculino. Ahora considere las consultas 1 y 2.*

1.
```
WITH ( STATS WHERE SEXO ■ 'M' AND
       OCUPACIÓN • 'Programado:-1 ) AS X :
COUNT ( X )
```

   Resultado: 1.

2.
```
WITH ( STATS WHERE SEXO - 'M' AND
       OCUPACIÓN = 'Programador1 ) AS X :
SUM ( X, SALARIO )
```

   Resultado: 50K. |

La seguridad de la base de datos ha sido comprometida claramente; aun cuando el usuario *U* sólo ha emitido consultas estadísticas legítimas. Como ilustra el ejemplo, si el usuario puede encontrar una expresión lógica que identifica a algún individuo, entonces la información con relación a ese individuo ya no es segura. Este hecho sugiere que el sistema debe rehusarse a responder una consulta para la cual la cardinalidad del conjunto a resumir es menor que algún límite inferior *b.* En forma similar, sugiere que el sistema también debe rehusarse a responder si esa cardinalidad es mayor que el límite superior *N - b* (donde *N* es la cardinalidad de la relación contenedora), ya que el problema anterior también podría surgir a partir de la secuencia de las consultas 3 a 6 que están a continuación:

3.
```
COUNT ( STATS )
```

   Resultado: 12.

4.
```
WITH ( STATS WHERE NOT ( SEXO = 'M' AND
       OCUPACIÓN = 'Programador' ) ) AS X :
COUNT ( X )
```

   Resultado: 11; 12-11 = 1.

5.
```
SUM ( STATS, SALARIO )
```

   Resultado: 728K.

6.
```
WITH ( STATS WHERE NOT ( SEXO = 'M' AND
       OCUPACIÓN = 'Programador1 ) ) AS X :
SUM ( X, SALARIO )
```

   Resultado: 678K; 728K - 678K = 50K. |

Por desgracia, es fácil mostrar que, en general, restringir las consultas a aquellas para la cuales el conjunto a resumir tiene cardinalidad *c* en el rango *b* £ *c* £ *N* — *b,* es inadecuada par; evitar el problema. De nuevo considere la figura 16.2 y suponga que *b* = 2. Las consultas serái respondidas sólo si c está en el rango 2 ^ *c* £ 8. Por lo tanto, la expresión lógica

```
SEXO = 'M' AND OCUPACIÓN ■ 'Programador1
```

---

*Para ahorrar tiempo, tocias las consultas de esta sección están expresadas en la forma abreviada del Tute rial D. Por ejemplo, la expresión COUNT (X) en la consulta 1, debería ser escrita (en forma más adecuad; como EXTEND TABLEJDEE ADD COUNT(X) AS RESULT1.

ya no es admisible. Pero considere la siguiente secuencia de consultas 7 a 10:

```
7. WITH ( STATS WHERE SEXO = 'M' ) AS X :
   COUNT ( X )
```

Resultado: 4.

```
8. WITH ( STATS WHERE SEXO = 'M' AND NOT
                  ( OCUPACIÓN =  'Programador¹  ) ) AS X :
   COUNT ( X )
```

Resultado: 3.

A partir de las consultas 7 y 8, el usuario *U* puede deducir que existe solamente un programador de sexo masculino, quien por lo tanto debe ser Alf (ya que *U* sabe que esta descripción corresponde a Alf). Entonces, el salario de Alf puede ser descubierto de la siguiente forma:

```
9. WITH ( STATS WHERE SEXO = 'M' ) AS X :
   SUM ( X, SALARIO )
```

Resultado: 328K.

```
10. WITH ( STATS WHERE SEXO • 'M' AND NOT
                   ( OCUPACIÓN « 'Programador¹ ) ) AS X :
    SUM ( X, SALARIO )
```

Resultado: 278K; 328K - 278K = 50K. |

La expresión lógica SEXO = 'M' AND OCUPACIÓN = 'Programador' se llama **rastreador individual** para Alf [16.6], ya que permite que el usuario siga la pista de la información relacionada con Alf. En general, si el usuario conoce una expresión lógica *BE* que identifica a algún individuo específico /, y si *BE* puede ser expresada en la forma *BE1* AND *BE2,* entonces la expresión lógica *BE1* AND NOT *BE2* es un rastreador para / (siempre y cuando tanto *BE1* como *BE1* AND NOT *BE2* sean admisibles; es decir, que ambas identifiquen a conjuntos de resultados que tengan cardinalidad *c* en el rango í>ScSJV-fc).La razón es que el conjunto identificado por *BE* es idéntico a la diferencia entre el conjunto identificado por *BE1* y el conjunto identificado por *BE1* AND NOT *BE2:*

```
conjunto ( BE ) = conjunto ( BE1 AND BE2 )
                • conjunto ( BE1 ) menos conjunto ( BE1 AND NOT BE2 )
```

Vea la figura 16.3.

La referencia [16.6] generaliza las ideas anteriores y muestra que para *casi cualquier* base de datos estadística, siempre es posible encontrar un **rastreador general** (al contrario de un conjunto de rastreadores individuales). Un rastreador general es una expresión lógica que puede ser usada para encontrar la respuesta de *cualquier* consulta inadmisible; es decir, cualquier pregunta que involucre una expresión inadmisible. (Por el contrario, un rastreador individual funciona solamente para consultas que involucren alguna expresión inadmisible *específica.)* De hecho, cualquier expresión cuya cardinalidad del conjunto resultante sea *c* dentro del rango *2b^c^N-2b,* es un rastreador general *(b* debe ser menor que *NÍA,* lo cual ocurrirá típicamente en cualquier situación realista). Una vez encontrado un rastreador de este tipo, es posible responder una consulta que involucra una expresión inadmisible de *BE,* como ilustra el siguiente ejemplo. (Para concretar, consideraremos el caso donde la cardinalidad del conjunto resultante correspondiente

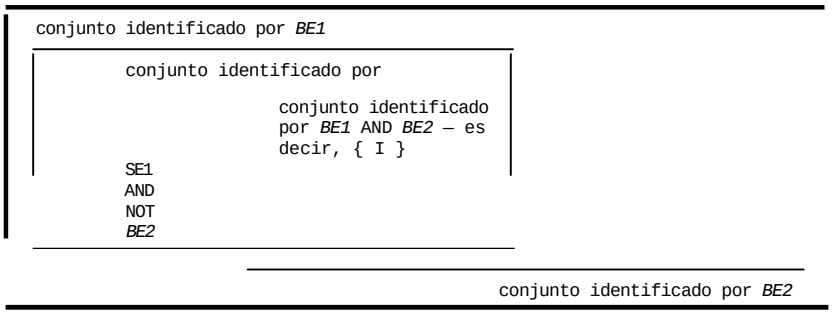


Figura 16.3 El rastreador individual *BE1* AND NOT BE2.

a *BE* es menor que *b.* En forma similar se maneja el caso donde es mayor que *N-b.)* Observe que de la definición podemos deducir que Tes un rastreador general si y sólo si NOT *T*también es un rastreador general.

Ejemplo: De nuevo supongamos que $b = 2$; entonces, un rastreador general es cualquier expresión que tenga un conjunto resultante con cardinalidad *c* en el rango $4 < c \wedge 6$. Supongamos nuevamente que el usuario *U* sabe por fuentes externas que Alf es un programador de sexo masculino; es decir, la expresión lógica inadmisible *BE* es (igual que antes)

```
SEXO ■ 'M' AND OCUPACIÓN = 'Programador¹
```

Supongamos también que *U* desea descubrir el salario de Alf. Usaremos dos veces un rastreador general, primero para asegurarnos que *BE* en efecto identifique a Alf en forma única (pasos 2 a 4) y luego para determinar el salario de Alf (pasos 5 a 7).

Paso 1: Haga una suposición con el rastreador *T.* Para nuestra suposición decidimos que Tes la expresión:

```
AUDITORIAS = 0
```

Paso 2: Obtenga la cantidad total de individuos en la base de datos usando las expresiones TyNOTr.

```
WITH ( STATS WHERE AUDITORIAS = 0 ) AS X :
COUNT ( X )
```

Resultado: 5.

```
WITH ( STATS WHERE NOT ( AUDITORIAS = 0 ) ) AS X :
COUNT ( X )
```

Resultado: 5; 5 + 5 = 10.

Ahora puede ver fácilmente que nuestra suposición *T* sí es un rastreador general.

Paso 3: Obtenga el resultado de sumar (a) la cantidad de individuos de la base de datos más (b) la cantidad que satisface la expresión inadmisible *BE* usando las expresiones *BE* OR *Ty BE* OR NOT *T.*

```
WITH ( STATS WHERE ( SEXO = 'M' AND
                     OCUPACIÓN = 'Programador' )
OR AUDITORIAS = 0 ) AS X : COUNT ( X )
```

Resultado: 6.

```
WITH ( STATS WHERE ( SEXO = 'M' AND
                     OCUPACIÓN = 'Programador¹ )
OR NOT ( AUDITORIAS = 0 ) ) AS X : COUNT ( X )
```

Resultado: 5; 6 + 5 = 11.

Paso 4: A partir de los resultados obtenidos, tenemos que la cantidad de individuos que satisface a *BE* es 1 (el resultado del paso 3 menos el resultado del paso 2); es decir, *BE* designa a Alf en forma única.

Ahora repetimos (en los pasos 5 y 6) las consultas de los pasos 2 y 3, pero usando SUM en lugar de COUNT.

Paso 5: Obtenga el salario total de los individuos en la base de datos, usando las expresiones *T* y NOT *T*.

```
WITH ( STATS WHERE AUDITORIAS = 0 ) AS X :
SUM ( X, SALARIO )
```

Resultado: 43 8K.

```
WITH ( STATS WHERE NOT ( AUDITORIAS = 0 ) ) AS X :
SUM ( X, SALARIO )
```

Resultado: 290K; 438K + 290K = 728K.

Paso 6: Obtenga la suma del salario de Alf y el salario total usando las expresiones *BE* OR *T* y *BE* OR NOT *T*.

```
WITH ( STATS WHERE ( SEXO = 'M' AND
                     OCUPACIÓN • 'Programador¹ ) OR
AUDITORIAS = 0 ) AS X : SUM ( X, SALARIO )
```

Resultado: 488K.

```
WITH ( STATS WHERE ( SEXO = 'M' AND
                     OCUPACIÓN = 'Programador' )
OR NOT ( AUDITORIAS = 0 ) ) AS X : SUM ( X,
SALARIO )
```

Resultado: 290K; 488K + 290K = 778K.

Paso 7: Obtenga el salario de Alf restando el salario total (que se encontró en el paso 5) del resultado del paso 6. *Resultado:* 50 K. |

La figura 16.4 ilustra al rastreador general:

```
conjunto ( BE ) = ( conjunto ( BE OR T ) más conjunto ( BE OR NOT r ) )
                                              menos conjunto ( T OR NOT T )
```

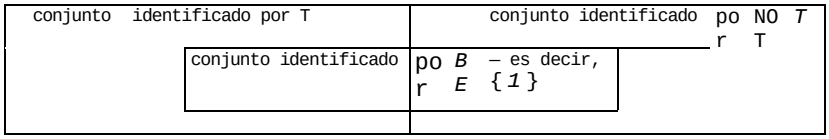


Figura 16.4 El rastreador general T.

Si la suposición inicial estuvo equivocada (es decir, si resulta que *Tno* es un rastreador general), entonces una o ambas expresiones *(BE* OR *T)* y *(BE* OR NOT 7) pueden ser inadmisibles. Por ejemplo, si las cardinalidades del conjunto resultante para *BE* y *Tsonp* y *q,* respectivamente (donde $p < b$ y $b < q < 2b$) entonces es posible que la cardinalidad del conjunto resultante para *(BE* OR 7) —la cual no puede exceder a $p + q$— sea menor que 2b. En tal situación es necesario hacer otra suposición con el rastreador y volverlo a intentar. La referencia [16.6] sugiere que el proceso de encontrar un rastreador general no es difícil en la práctica. En nuestro ejemplo particular, la suposición inicial *es* un rastreador general (la cardinalidad en su conjunto resultante es 5) y ambas consultas del paso 3 son admisibles.

En resumen, "casi siempre" existe un rastreador general y por lo regular es fácil encontrarlo y usarlo; de hecho, a menudo es posible encontrar rápidamente un rastreador por mera suposición [16.6]. Incluso en los casos donde no existe un rastreador general, la referencia [16.6] muestra que (por lo general) es posible encontrar rastreadores específicos para consultas específicas. Es difícil no concluir que la seguridad en una base de datos estadística es un problema real.

¿Qué podemos hacer entonces? Se han hecho diversas sugerencias en la literatura, pero no es claro que alguna de ellas sea totalmente satisfactoria. Por ejemplo, una posibilidad es el "intercambio de datos"; es decir, el intercambio de valores de atributos entre tupias en una forma tal que mantenga la precisión estadística, de manera que aunque se identifique un valor en particular (digamos, un salario específico), no haya forma de saber a qué individuo en especial corresponde. La dificultad de este enfoque radica en encontrar conjuntos de entradas cuyos valores puedan ser intercambiados de esta forma. Otras limitaciones similares se aplican a la mayoría de las soluciones sugeridas. Entonces, por el momento parece que debemos aceptar las conclusiones de Denning [16.6]: "El compromiso es directo y barato. El requerimiento de mantenerla información confidencial en completo secreto no coincide con el requerimiento de producir medidas estadísticas exactas para subconjuntos arbitrarios de la población. Al menos uno de estos requerimientos debe ser relajado antes de que podamos creer que el secreto está seguro."

16.5 CIFRADO DE DATOS

En este capítulo hemos supuesto —hasta el momento— que cualquier posible infiltrador estará usando las facultades normales del sistema para acceder a la base de datos. Ahora pondremos nuestra atención en el caso de un "usuario" que trata de *dejar de lado* al sistema (por ejemplo, eliminando físicamente parte de la base de datos o interviniendo una línea de comunicación). La medida más efectiva en contra de tal amenaza es el **cifrado de datos** (o encriptación, como también se conoce); es decir, el guardado y la transmisión de datos sensibles en forma cifrada.

Para explicar algunos de los conceptos del cifrado de datos necesitamos presentar un poco más de terminología. A los datos originales (sin cifrado) se les llama **texto plano.** El texto plano es **cifrado** sometiéndolo a un **algoritmo de cifrado** —cuyas entradas son el texto plano y la **clave de cifrado**— y a la salida de este algoritmo —la forma cifrada del texto plano— se le llama texto **cifrado.** Los detalles del algoritmo de cifrado son públicos —o al menos no están ocultos especialmente— pero la clave de cifrado se mantiene en secreto. El texto cifrado, que debe ser ininteligible para cualquiera que no posea la clave de cifrado, es lo que se guarda en la base de datos o se transmite por la línea de comunicación.

Ejemplo: Hagamos que el texto plano sea la cadena:

```
AS KINGFISHERS CATCH FIRE
```

(Suponemos, por simplicidad, que los únicos caracteres de datos que tenemos que manejar son las letras mayúsculas y los espacios en blanco). Hagamos que la clave de cifrado sea la cadena

```
ELIOT
```

y que el algoritmo de cifrado sea el siguiente:

1. Dividimos el texto plano en bloques de longitud igual a la clave de cifrado:

   ```
   AS + KI   NGFIS   HERS+   CATCH   +FIRE
   ```

   (los espacios en blanco ahora son mostrados explícitamente como "+").

2. Reemplazamos cada carácter del texto plano por un entero que esté en el rango de 00 a 26, usando espacio en blanco = 00, A = 01,..., Z = 26:

   ```
   0119001109   1407060919   0805181900   0301200308   0006091805
   ```

3. Repetimos el paso 2 para la clave de cifrado:

   ```
   0512091520
   ```

4. Para cada bloque de texto plano reemplazamos cada carácter por la suma módulo 27 de su codificación de enteros más la codificación de enteros del carácter correspondiente de la clave de cifrado:

   ```
   0119001109  1407060919  0805181900  0301200308  0006091805
   0512091520  0512091520  0512091520  0512091520  0512091520
   ----------  ----------  ----------  ----------  ----------
   0604092602  1919152412  1317000720  0813021801  0518180625
   ```

5. Reemplazamos cada codificación de enteros del resultado del paso 4 por su equivalente en caracteres:

   ```
   FDIZB   SS0XL   MQ+GT   HMBRA   ERRFY
   ```

El procedimiento de descifrado para este ejemplo es directo, *siempre y cuando se tenga la clave. {Ejercicio:* Descifre el texto cifrado mostrado anteriormente.) La pregunta es, ¿qué tan difícil será para un posible infiltrador determinar la clave sin ningún conocimiento previo, teniendo el texto plano y el texto cifrado? En nuestro ejemplo la respuesta es, obviamente, "no mucho"; pero también es obvio que es posible inventar esquemas mucho más sofisticados. De

manera idónea, el esquema empleado debería ser tal, que el trabajo involucrado en romperlo sobrepasara cualquier ventaja potencial que pudiera obtenerse al hacerlo. (De hecho, la misma idea se aplica a todos los aspectos de la seguridad; el objetivo debe ser siempre hacer que el costo de la ruptura del sistema sea significativamente mayor que el beneficio potencial.) El objetivo último aceptado de tales esquemas es que el *inventor* del esquema, que tiene el texto plano y el texto cifrado correspondiente, debe ser incapaz de determinar la clave y por lo tanto, incapaz de descifrar otra parte de texto cifrado.

El estándar de cifrado de datos

El ejemplo anterior hace uso de un procedimiento de **sustitución:** se usa una clave de cifrado para determinar (para cada carácter del texto plano) un carácter de texto cifrado que va a *sustituir* a ese carácter. La sustitución es uno de los dos enfoques básicos del cifrado como se practica tradicionalmente; el otro es el de la **permutación,** donde los caracteres del texto plano son simplemente reorganizados en una secuencia diferente. Ninguno de estos enfoques es particularmente seguro en sí mismo, pero los algoritmos que combinan a los dos pueden proporcionar un alto grado de seguridad. Uno de estos algoritmos es el **estándar de cifrado de** datos (DES), desarrollado por IBM y adoptado como estándar federal de los Estados Unidos en 1977 [16.18].

Para usar el DES, el texto plano es dividido en bloques de 64 bits y cada bloque es cifrado usando una clave de 64 bits (de hecho, la clave consta de 56 bits de datos y 8 bits de paridad, por lo que las claves posibles no son 2^{64}sino sólo 2^{56}). Un bloque es cifrado aplicándole una permutación inicial, sometiéndolo posteriormente a una secuencia de 16 pasos complejos de sustitución y aplicando finalmente otra permutación —la inversa de la permutación inicial— al resultado del último de estos pasos. La sustitución en el ¡esimo paso no está controlada directamente por la clave de cifrado original *K*, sino por una clave *Ki* que es calculada a partir de los valores de *K* e *i*. Para mayores detalles vea la referencia [16.18].

El DES tiene la propiedad de que el algoritmo de descifrado es idéntico al algoritmo de cifrado, con la excepción de que las *Ki* se aplican en orden inverso.

Cifrado de clave pública

A través de los años muchas personas han sugerido que probablemente el DES en realidad no es seguro; de hecho, la aparición de procesadores muy rápidos y altamente paralelos sugiere que el DES puede ser roto por fuerza bruta (si no es que por medios más inteligentes). Muchos también piensan que los esquemas de cifrado de "clave pública" más recientes, dejan obsoletos tecnológicamente a los enfoques tradicionales DES y similares. En un esquema de clave pública, tanto el algoritmo de cifrado como la *clave de cifrado* están disponibles y por lo tanto, cualquier persona puede convertir texto plano en texto cifrado. Pero la **clave de descifrado** correspondiente se mantiene en secreto (los esquemas de clave pública involucran *dos* claves, una para el cifrado y otra para el descifrado). Además, la clave de descifrado no puede ser deducida de manera realista a partir de la clave de cifrado; por consiguiente, incluso la persona que realiza el cifrado original no puede realizar el descifrado correspondiente si no tiene autorización para ello.

La idea original del cifrado por clave pública se debe a Diffie y Hellman [16.7]. Describimos el enfoque específico más conocido —de Rivest, Shamir y Adleman [16.15]— para mostrar la manera en que el esquema funciona en la práctica. Su enfoque (al que se le conoce generalmente

como *esquema RSA,* debido a las iniciales de sus inventores) está basado en los dos siguientes hechos:

1. Existe un algoritmo rápido para determinar si un número dado es primo.
2. No existe ningún algoritmo rápido para encontrar los factores primos de un número com puesto (es decir, no primo) dado.

La referencia [16.10] proporciona un ejemplo en el que determinar (en una máquina dada) si un cierto número de 130 dígitos es primo, toma cerca de 7 minutos; mientras que encontrar los dos factores primos (en la misma máquina) de un número obtenido de multiplicar dos números primos de 63 dígitos se llevaría cerca de *40 mil billones* de años (mil billones = 1,000,000,000,000,000).*

El esquema RSA funciona de la siguiente manera:

1. Se seleccionan al azar dos números primos grandes, *p* y *q,* diferentes y se calcula el pro ducto *r = p*q.*
2. Se selecciona al azar un número entero grande, *e,* que sea primo relativo; es decir, que no tenga factores en común con respecto al producto *(p* - 1) * *(q* - 1). El número entero *e* es la clave de cifrado. *Nota:* La selección de *e* es directa, ya que podemos usar cualquier número primo que sea mayor ap y *q.*
3. Se toma la clave de descifrado, *d,* para que sea el "multiplicativo inverso" único de *e* mó dulo *(p* - 1) * *(q* -1), es decir,

   d * e = 1 módulo (p - 1) * (q - 1)

   El algoritmo para calcular *d* a partir de *e, p* y *q,* es directo y es proporcionado en la referencia [16.15].
4. Se publican los enteros rye, pero no *d.*
5. Para cifrar un texto plano *P* (por simplicidad, suponemos que es un entero menor que *r)* lo reemplazamos por el texto cifrado C, el cual se calcula de la siguiente manera:

   C = P" módulo r

6. Para descifrar una parte del texto cifrado *C* lo reemplazamos por el texto plano *P,* que se calcula de la manera siguiente:

   P = C módulo *r*

La referencia [16.15] prueba que este esquema funciona; es decir, que el descifrado de *C* usando *d* recupera el *P* original. Sin embargo, como afirmamos anteriormente, calcular *d* conociendo únicamente *r* y *e* (y no *p* o *q)* no es factible. Por lo tanto, cualquier persona puede cifrar texto plano, pero sólo los usuarios autorizados (que tienen *d)* pueden descifrar el texto cifrado.

---

*A pesar de ello, existen algunas dudas acerca de la seguridad del esquema RSA. La referencia [16.10] fue publicada en 1977. En 1990 Arjen Lenstra y Mark Manasse factorizaron satisfactoriamente un número de 155 dígitos [16.22] y estimaron que el tiempo de cálculo involucrado (que fue repartido en mil computadoras) era equivalente a ejecutar un millón de instrucciones por segundo en una sola máquina durante 273 años. El número de 155 dígitos fue el noveno número de Fermat $2^{512}+ 1$ (observe que $512 = 2^9$). Vea también la referencia [16.12], la cual reporta un enfoque completamente diferente ¡y exitoso! para la ruptura del cifrado RSA.

Damos un ejemplo trivial para ilustrar el procedimiento anterior. Por razones obvias nos limitamos a números muy pequeños.

Ejemplo: Hagamos*p* = 3, *q* = *5;* entonces r= 15 y el producto *(p* - 1) * *(q*- 1) = 8. Hagamos *e* = 11 (un número primo mayor que *p* y *q).* Para calcular *d* tenemos

```
d * 11 =1 módulo 8
```

y por lo tanto, *d* = 3.

Ahora hagamos que el texto plano *P* consista en el entero 13. Entonces, el texto cifrado *C* está dado por

```
C = f módulo r = 13" módulo 15
  = 1,792,160,394,037 módulo 15
  ■ 7
```

Ahora el texto plano *P* original está dado por

```
P = C módulo r
  ■ 7^3 módulo 15
  = 343 módulo 15
  - 13 I
```

Puesto que *e* y *d* son inversos entre sí, los esquemas de cifrado de clave pública también permiten que los mensajes cifrados sean **"firmados"** en forma tal que el receptor pueda estar seguro de que el mensaje se originó en la persona que dice haberlo hecho (es decir, las "firmas" no se pueden falsificar). Supongamos que *A* y *B* son dos usuarios que desean comunicarse entre sí usando un esquema de cifrado de clave pública. Entonces *Ay B* publicarán un algoritmo de cifrado (incluyendo en cada caso la clave de cifrado correspondiente) pero por supuesto, mantendrán el algoritmo de descifrado y la clave en secreto (incluso entre sí). Hagamos que los algoritmos de cifrado sean ECA y ECB (para cifrar mensajes que serán enviados a *A* y *B,* respectivamente) y hagamos que los algoritmos de descifrado correspondientes sean DC A y DCB, respectivamente. ECA y DC A son inversos entre sí, al igual que ECB y DCB.

Ahora supongamos que *A* desea enviar a *B* un fragmento de texto plano *P.* En lugar de calcular ECB *(P)* y transmitir el resultado, *A* aplica primero a *P* el algoritmo de *descifrado* DCA y luego cifra el resultado y lo transmite como el texto cifrado C:

```
C = ECB ( DCA ( P ) )
```

Al recibir C, el usuario *B* aplica el algoritmo de descifrado DCB y luego el algoritmo de *cifrado* ECA produciendo el resultado final *P:*

```
ECA ( DCB ( C ) )
  = ECA ( DCB ( ECB  ( DCA ( P ) ) ) )
  ■ ECA ( DCA ( P )              ) I* ya    que DCB y ECB se
  cancelan */
  = P     /* ya que ECA y DCA se cancelan */
```

Ahora *B* sabe que el mensaje en efecto proviene de *A,* ya que ECA producirá *P* sólo si el algoritmo DCA fue utilizado en el proceso de cifrado y ese algoritmo sólo lo conoce a A. Nadie, *incluso B,* puede falsificar la firma de *A.*

6.6 PROPIEDADES DE SQL

El estándar actual de SQL soporta únicamente el control de acceso discrecional. Hay dos características (más o menos independientes de SQL) involucradas; éstas son: el **mecanismo de vistas,** el cual (como sugerí en el capítulo 8) puede ser usado para ocultar datos sensibles ante usuarios no autorizados, y el **subsistema de autorización,** el cual permite a los usuarios con privilegios específicos otorgar en forma selectiva y dinámica esos privilegios a otros usuarios y posteriormente —de ser necesario— revocar esos privilegios. Ambas características son explicadas a continuación.

Las vistas y la seguridad

Para ilustrar el uso de las vistas con propósitos de seguridad en SQL, primero damos las versiones de SQL equivalentes a los ejemplos de vistas (ejemplos 2 al 4) de la sección 16.2.

2.
```
CREATE VIEW VL AS
       SELECT V.V#, V.PROVEEDOR, V.STATUS, V.CIUDAD
       FROM   V
       WHERE V,CIUDAD = 'Londres' ;
```

La vista define los datos sobre los cuales va a ser otorgada la autorización. El otorgamiento se realiza por medio de la instrucción GRANT, por ejemplo:

```
GRANT SELECT, UPDATE ( PROVEEDOR, STATUS ), DELETE
ON    VL
TO    Dan, Misha ;
```

Observe que —puesto que están definidas por medio de una instrucción GRANT especial (como se muestra), en lugar de una instrucción "CREATE AUTHORITY" hipotética— las autoridades son *anónimas* en SQL. (Por el contrario, las restricciones de integridad sí tienen nombres, como vimos en el capítulo 8.)

3.
```
CREATE VIEW VVPPR AS
       SELECT V.V#, V.PROVEEDOR, V.STATUS, V.CIUDAD
       FROM   V WHERE EXISTS
            ( SELECT * FROM VP
              WHERE EXISTS
                   ( SELECT * FROM P WHERE V.V# •
                     VP.V# AND   VP.P# = P.P# AND
                     P.CIUDAD = 'Roma' ) ) ;
```

La instrucción GRANT correspondiente es:

```
GRANT SELECT ON VVPPR TO Giovanni ;
```

4.
```
CREATE VIEW VVC AS
       SELECT V.V#, ( SELECT SUM ( VP.CANT )
                      FROM VP
                      WHERE VP.V# = V.V# ) AS VC
       FROM V ;
```

La instrucción GRANT correspondiente es:

```
GRANT SELECT ON VVC TO Fidel ;
```

El ejemplo 5 de la sección 16.2 involucra a una autoridad *dependiente del contexto.* SQL soporta una variedad de operadores niládicos integrados, CURRENT_USER, CURRENT_DATE, CURRENTJTIME, etcétera, que pueden ser usados —entre otras cosas— para definir las vistas dependientes del contexto (sin embargo, observe que SQL no soporta algo similar al operador DAY() que usamos en nuestro ejemplo 5 original). He aquí un ejemplo:

```
CREATE VIEW V_NUEVE_A_CINCO AS
       SELECT V.V#, V.PROVEEDOR, V.STATUS, V.CIUDAD
       FROM   V
       WHERE CURRENT_TIME > TIME '09:00:00'
       AND   CURRENTJTIME < TIME '17:00:00' ;
```

La instrucción GRANT correspondiente es:

```
GRANT SELECT, UPDATE ( STATUS )
ON    VNUEVEACINCO TO
Compras ;
```

Sin embargo, observe que V_NUEVE_A_CINCO es un tipo de vista extraño, ya que su valor cambia a través del tiempo aunque sus datos subyacentes nunca cambian. Además, una vista cuya definición involucra al operador integrado CURRENTJJSER puede (de hecho, es probable que así sea) tener diferentes valores aun para diferentes usuarios. Dichas "vistas" son en realidad diferentes con respecto a las vistas como normalmente se entienden, ya que en realidad están *parametrizadas.* Tal vez sería preferible (al menos conceptualmente) permitir que los usuarios definieran sus propias funciones evaluadas por relación —potencialmente parametrizadas— y luego trataran a las "vistas" tipo V_NUEVE_A_CINCO como casos especiales de dichas funciones.

Siendo así, los ejemplos anteriores ilustran la idea de que el mecanismo de vistas proporciona una medida de seguridad importante "gratuita" ("gratuita" ya que el mecanismo fue incluido en el sistema con otros propósitos). Lo que es más, muchas revisiones de autorización —incluso las que dependen de valores— pueden ser realizadas en tiempo de compilación en lugar de realizarlas en tiempo de ejecución, lo que implica un beneficio importante en el desempeño. Sin embargo, en ocasiones el enfoque de la seguridad basado en vistas enfrenta algunas dificultades ligeras; en especial si algún usuario específico necesita al mismo tiempo privilegios diferentes sobre distintos conjuntos de la misma tabla. Por ejemplo, considere la estructura de una aplicación a la que se le permite revisar e imprimir todas las partes de Londres y también actualizar alguna de ellas (digamos que sólo las rojas) durante la revisión.

GRANT y REVOKE

El mecanismo de vistas permite dividir conceptualmente a la base de datos en partes de diversas formas para que la información delicada pueda ocultarse de usuarios no autorizados. Sin embargo, no permite especificar las operaciones que los usuarios *autorizados* pueden ejecutar sobre esas partes. Esa tarea (como ya hemos visto en los ejemplos anteriores) es realizada por la instrucción **GRANT,** la cual trataremos ahora con más detalle.

Observe primero que al creador de cualquier objeto se le otorgan automáticamente todos los privilegios que tienen sentido para ese objeto. Por ejemplo, al creador de una tabla base *T* se le otorgan automáticamente privilegios SELECT, INSERT, UPDATE, DELETE y REFERENCES sobre *T* (más adelante encontrará una explicación de estos privilegios). Además, estos privilegios

son otorgados en cada caso "con autoridad de otorgamiento", lo que significa que el usuario que los posee puede otorgarlos a otros usuarios.

Ésta es la sintaxis de la instrucción GRANT:

```
GRANT <lista de privilegios separados con comas> ON
      <objeto>
      TO <lista de IDs de usuario separados con comas> [
    WITH GRANT OPTION]  ;
```

Explicación:

- Los <*privilegios*> válidos son USAGE, SELECT, INSERT, UPDATE, DELETE y REFE RENCES.* El privilegio USAGE es necesario para usar un dominio específico (estilo SQL); el privilegio REFERENCES es necesario para hacer referencia a una tabla con nombre específico en una restricción de integridad; los demás privilegios se entienden por sí mis mos. Sin embargo, observe que los privilegios de INSERT, UPDATE y REFERENCES (aunque extrañamente, no el privilegio de SELECT) pueden ser específicos de la columna. *Nota:* También es posible especificar ALL PRIVILEGES, pero la semántica no es tan directa [4.19].
- Los <*objetos*> válidos son DOMAIN *-oiombre de dominie»* y TABLE <*nombre de tabla*>. *Nota:* En este contexto, a diferencia de muchos otros en SQL, la palabra reservada "TABLE" (que de hecho es opcional) incluye tanto a las vistas como a las tablas base.
- La <*lista de IDs de usuario separados con comas*> puede ser reemplazada por la pala bra reservada especial PUBLIC, con el significado: "todos los usuarios conocidos por el sistema".
- Si WITH GRANT OPTION está especificado, significa que a los usuarios especificados se les otorgan los privilegios especificados sobre el objeto especificado **con autoridad de otorgamiento;** lo que significa, como dijimos antes, que es posible otorgar esos privilegios sobre ese objeto a algún otro usuario. Por supuesto, WITH GRANT OPTION sólo puede ser especificado cuando el usuario que emite la instrucción GRANT tiene en primer lugar la autoridad de otorgamiento necesaria.

Entonces, si un usuario *A* otorga algún privilegio a algún otro usuario *B,* el usuario *A* puede *revocar* posteriormente ese privilegio al usuario *B.* La revocación de privilegios se realiza por medio de la instrucción **REVOKE,** y ésta es su sintaxis:

```
REVOKE [ GRANT OPTION FOR ] <lista de privilegios separados con comas>
                            ON <objeto>
                            FROM <lista de IDs de usuario separados con comas>
                            <opción> ;
```

Aquí (a) GRANT OPTION FOR significa que (sólo) se va a revocar la autoridad de otor-gamiento, (b) <*lista de privilegios separados con comas*>, <*objeto*> y <*lista de IDs de usuario separados con comas*> son similares a lo que se especificó para GRANT y (c) <*opción*> es RESTRICT o CASCADE, como explicaremos más adelante. Ejemplos:

1. REVOKE SELECT ON V FROM Jacques, Anne, Charley RESTRICT ;

*A1 agregar la característica de módulos almacenados persistentes (PSM) al estándar en 1996 [4.22], fue añadido un privilegio EXECUTE.

```
2. REVOKE SELECT, UPDATE ( PROVEEDOR, STATUS ), DELETE
          ON VL FROM Dan, Misha CASCADE ;

3. REVOKE SELECT ON VVPPR FROM Giovanni RESTRICT ;

4. REVOKE SELECT ON VVC FROM Fidel RESTRICT ;
```

RESTRICT vs. CASCADE: supongamos que/? es algún privilegio sobre algún objeto y supongamos que el usuario *A* otorga *p* al usuario *B,* quien a su vez lo otorga al usuario C. ¿Qué deberá pasar si *A* revoca ahora *p* a S? Supongamos por un momento que REVOKE tiene éxito. Entonces el privilegio *p* que tiene C quedaría "abandonado"; sería derivado de un usuario, como *B,* quien ya no lo tiene. El propósito de la opción RESTRICT *vs.* CASCADE es evitar la posibilidad de privilegios abandonados. Para ser más específicos, RESTRICT ocasiona que REVOKE falle si conduce a algún privilegio abandonado, y CASCADE ocasiona que tales privilegios sean revocados.

Por último, la eliminación de un dominio, tabla base, columna o vista revoca automáticamente todos los privilegios que tengan todos los usuarios sobre los objetos eliminados.

16.7 RESUMEN

Hemos explicado diversos aspectos del problema de la **seguridad** de la base de datos. Comenzamos contrastando la seguridad y la *integridad:* la seguridad involucra asegurar que a los usuarios se les permita hacer las cosas que están tratando de hacer y la integridad involucra asegurar que las cosas que dichos usuarios están tratando de hacer sean correctas. En otras palabras, la seguridad *involucra la protección de los datos contra su revelación, alteración o destrucción no autorizadas.*

La seguridad se hace cumplir mediante el **subsistema de seguridad** del DBMS, el cual verifica todas las peticiones de acceso contra las **restricciones de seguridad** almacenadas en el catálogo del sistema. Primero consideramos los esquemas **discrecionales,** donde el acceso a un objeto dado queda a la discreción del propietario del objeto. Cada **autoridad** en un esquema discrecional tiene un **nombre,** un conjunto de **privilegios** (RETRIEVE, UPDATE, etcétera), una **varrel** correspondiente (es decir, los datos a los que se aplica la restricción) y un conjunto de **usuarios.** Dichas autoridades pueden ser usadas para proporcionar controles **dependientes del valor, independientes del valor,** de **resúmenes estadísticos y dependientes del contexto.** Es posible usar un **registro de auditoría** para registrar los intentos de ruptura de la seguridad. Dimos un breve vistazo a una técnica de implementación para los esquemas discrecionales, conocida como **modificación de petición.** Esta técnica fue lanzada por primera vez en el prototipo de Ingres junto con el lenguaje QUEL.

Después tratamos los controles **obligatorios,** donde cada objeto tiene un nivel de **clasificación** y cada usuario tiene un nivel de **acreditación.** Explicamos las reglas para el acceso bajo este esquema. También resumimos el esquema de clasificación de seguridad definido por el Departamento de Defensa de los Estados Unidos en las referencias [16.19 y 16.20] y tratamos brevemente las ideas de **varrels de múltiples niveles y polinstanciación.**

Posteriormente tratamos los problemas especiales de las **bases de datos estadísticas.** Una base de datos estadística es aquella que contiene gran cantidad de conceptos de información individual sensible, pero que en teoría proporciona a sus usuarios solamente información de resúmenes estadísticos. Vimos que la seguridad de estas bases de datos es comprometida fácilmente por medio de los **rastreadores** (un hecho que debe causar un poco de alarma, tomando en cuen-

ta que el nivel de interés es cada vez mayor en los sistemas de almacenamiento de datos; vea el capítulo 21).

Luego examinamos el **cifrado de datos** y mencionamos las ideas básicas de **sustitución y permutación,** explicando lo que es el **estándar de cifrado de datos** y describiendo a grandes rasgos la forma en que funcionan los sistemas de **clave pública.** En particular, vimos un ejemplo simple del esquema **RSA** (número primo). También tratamos el concepto de **firma digital.**

También describimos brevemente las características de seguridad de SQL; en particular el uso de **vistas** para ocultar información y el uso de **GRANT y REVOKE** para controlar qué usuarios tienen qué privilegios sobre qué objetos (principalmente tablas base y vistas).

En conclusión, vale la pena enfatizar el punto de que no es bueno que el DBMS proporcione un amplio conjunto de controles de seguridad, si permite que esos controles sean ignorados de alguna forma. En DB2, por ejemplo, la base de datos está guardada físicamente como archivos del sistema operativo y por lo tanto, el mecanismo de seguridad del DB2 sería casi inútil si fuera posible acceder a esos archivos desde un programa convencional por medio de los servicios convencionales del sistema operativo. Por está razón, el DB2 trabaja en armonía con sus diversos sistemas integrados —en particular con el sistema operativo subyacente— para garantizar que el sistema total sea seguro. Los detalles están más allá del alcance de este capítulo, pero debe quedar claro el mensaje.

EJERCICIOS

16.1 Hagamos que la varrel base STATS sea como en la sección 16.4:

```
STATS { NOMBRE, SEXO, HIJOS, OCUPACIÓN, SALARIO, IMPUESTOS, AUDITORIAS }
      PRIMARY KEY { NOMBRE }
```

Usando el lenguaje hipotético presentado en la sección 16.2, defina las restricciones de seguridad necesarias para otorgar:

a. Al usuario Ford: privilegios de RETRIEVE sobre toda la varrel.
b. Al usuario Smith: privilegios de INSERT y DELETE sobre toda la varrel.
c. A cada usuario: privilegios de RETRIEVE sobre la tupia propia del usuario (solamente).
d. Al usuario Nash: privilegios de RETRIEVE sobre toda la varrel y privilegios de UPDATE sobre los atributos SALARIO e IMPUESTOS (solamente).
e. Al usuario Todd: privilegios de RETRIEVE sobre los atributos NOMBRE, SALARIO e IM PUESTOS (solamente).
f. Al usuario Ward: privilegios de RETRIEVE similares a los de Todd y privilegios de UPDATE sobre los atributos SALARIO e IMPUESTOS (solamente).
g. Al usuario Pope: privilegios completos (RETRIEVE, UPDATE, INSERT, DELETE) sobre tu pias de predicadores (solamente).
h. Al usuario Jones: privilegios de DELETE sobre tupias de personas que no tienen una ocupación especializada; donde la *ocupación no especializada* se define como aquella que pertenece a más de diez personas.
i. Al usuario King: privilegios de RETRIEVE para los salarios máximo y mínimo por ocupación.

16.2 Considere lo que implica extender la sintaxis de las instrucciones AUTHORITY para incluir un control sobre operaciones como definición y eliminación de varrels base, definición y eliminación de vistas, definición y eliminación de autoridades, etcétera.

16.3 Considere nuevamente la figura 16.2. Suponga que sabemos por otras fuentes que Hal es un cons tructor de casas con al menos dos hijos. Escriba una secuencia de consultas estadísticas que revela rán la cifra de impuestos de Hal usando un rastreador individual. Suponga, al igual que en la sección 16.4, que el sistema no responderá a consultas que tengan una cardinalidad del conjunto resultante menor que 2 o mayor que 8.

16.4 Repita el ejercicio 16.3, pero usando un rastreador general en lugar de uno individual.

16.5 Descifre el siguiente texto cifrado, que fue producido en una forma similar a la que usamos en el ejemplo "AS KINGFISHERS CATCH FIRE" de la sección 16.5, pero usando una clave de cifrado diferente de cinco caracteres:

```
F NWA L
J PVJ C
F PEX E
A BWN E
A YEI P
S USV D
```

16.6 Trabaje en el esquema de cifrado de clave pública RSA con $p = l, q = 5$ y $e = \backslash l$ para el texto plano $P = 3$.

16.7 ¿Puede imaginar cualquier problema de implementación u otra desventaja que pueda ser cau sada por el cifrado?

16.8 Dé soluciones SQL para el ejercicio 16.1.

16.9 Escriba instrucciones SQL para eliminar los privilegios otorgados en su solución al ejercicio anterior.

REFERENCIAS Y BIBLIOGRAFÍA

Para un panorama más amplio sobre la seguridad en general vea el libro de Castaño *et al.* [16.2]. Para un tratamiento técnico más detallado vea el libro de Denning [16.5]. Las demás referencias son en su mayoría, documentos de estándares o artículos técnicos (tutoriales o contribuciones de investigación) sobre diversos aspectos específicos del problema de la seguridad; también hay algunos artículos de periódicos.

16.1 D. E. Bell y L. J. La Padula: "Secure Computer Systems: Mathematical Foundations and Model", MITRE Technical Report M74-244 (mayo, 1974).

16.2 Silvana Castaño, Mariagrazia Fugini, Giancarlo Martella y Pierangela Samarati: *Database Se curity*. New York, N.Y.: ACM Press/Reading, Mass.: Addison-Wesley (1995).

16.3 James Daly: "Fingerprinting a Computer Security Code", *Computer-world* (julio 27, 1992).

16.4 C. J. Date: "Security", capítulo 4 *de An Introduction to Database Systems: Volume II.* Reading, Mass.: Addison-Wesley (1983).

16.5 Dorothy E. Denning: *Cryptography and Data Security*. Reading, Mass.: Addison-Wesley (1983).

16.6 Dorothy E. Denning y Peter J. Denning: "Data Security", *ACM Comp. Surv. 11,* No. 3 (septiem bre, 1979).

Es un buen tutorial que trata los controles de acceso discrecional, los controles de acceso obligatorio (llamados aquí controles *de flujo),* el cifrado de datos y los controles de *inferencia* (el problema especial de las bases de datos estadísticas).

16.7 W. Diffie and M. E. Hellman: "New Directions in Cryptography", *IEEE Transactions on In formation Theory* IT-22 (noviembre, 1976).

16.8 Ronald Fagin: "On an Authorization Mechanism", *ACM TODS 3,* No. 3 (septiembre, 1978).

Una corrección amplia a la referencia [16.11]. Bajo ciertas circunstancias el mecanismo de la referencia [16.11] eliminaría un privilegio que no debería ser eliminado. Este artículo corrige esa falla.

16.9 Roberto Gagliardi, George Lapis y Bruce Lindsay: "A Flexible and Efficient Database Autho rization Facility", IBM Research Report RJ6826 (mayo 11, 1989).

16.10 Martin Gardner: "A New Kind of Cipher That Would Take Millions of Years to Break", *Sci entific American 237,* No. 2 (agosto, 1977).

Es una buena introducción informal al trabajo realizado sobre el cifrado de clave pública. El título puede ser una exageración [16.12], [16.22].

16.11 Patricia P. Griffiths y Bradford W. Wade: "An Authorization Mechanism for a Relational Data Base System", *ACM TODS 1,* No. 3 (septiembre, 1976).

Describe el mecanismo de GRANT y REVOKE propuesto originalmente para el System R. El esquema que está actualmente en el estándar SQL está basado en ese mecanismo, aunque los detalles son muy diferentes.

16.12 Nigel Hawkes: "Breaking into the Internet", *London Times* (marzo 18, 1996).

Describe la manera en que un experto en computación rompió el esquema RSA midiendo cuánto tiempo necesita el sistema para descifrar mensajes, "el equivalente electrónico de adivinar la combinación de una caja fuerte observando a alguien girar la perilla y viendo qué tanto lo hace".

16.13 Sushil Jajodia y Ravi Sandhu: "Toward a Multi-Level Secure Relational Data Model", Proc. 1991 ACM SIGMOD Int. Conf. on Management of Data, Denver, Colo, (junio, 1991).

Como expliqué en la sección 16.3, "múltiples niveles" hace referencia —en un contexto de seguridad— a un sistema que soporta controles de acceso obligatorios. Este artículo sugiere que gran parte de la actividad actual en el campo es hecha *a la medida* debido a que hay muy poco consenso sobre los conceptos básicos y propone un inicio formalizando los principios de los sistemas de múltiples niveles.

16.14 Abraham Lempel: "Cryptology in Transition", *ACM Comp. Surv. 11,* No. 4: Special Issue on Cryptology (diciembre, 1979).

Es un buen tutorial sobre el cifrado y algunos temas relacionados.

16.15 R. L. Rivest, A. Shamir y L. Adleman: "A Method for Obtaining Digital Signatures and Pub lic Key Cryptosystems", *CACM21,* No. 2 (febrero, 1978).

16.16 Ken Smith y Marianne Winslett: "Entity Modeling in the MLS Relational Model", Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (agosto, 1992).

En el título de este artículo, "MLS" significa "seguro de múltiples niveles" [16.13], Este artículo se enfoca en el *significado* de las bases de datos MLS y propone una nueva cláusula BELIEVED BY sobre las operaciones de recuperación y actualización, con el fin de dirigir estas operaciones hacia el estado particular de la base de datos que es entendido o "creído" por un usuario específico. Se dice que este enfoque resuelve varios problemas que se presentan con enfoques anteriores. Vea también la referencia [16.21].

16.17 Bhavani Thuraisingham: "Current Status of R&D in Trusted Database Management Systems", *ACM SIGMOD Record 21,* No. 3 (septiembre, 1992).

Es un breve estudio y un amplio conjunto de referencias sobre sistemas "confiables" o de múltiples niveles (como era a principios de los años noventa).

16.18 U.S. Department of Commerce /National Bureau of Standards: *Data Encryption Standard.* Federal Information Processing Standards Publication 46 (enero 15, 1977).

Define el estándar oficial de cifrado de datos (DES) a ser usado por las agencias federales y cualquier otra persona que lo desee. Es adecuado implementar el algoritmo de cifrado/descifrado (vea la sección 16.5) en un chip de hardware; lo que significa que los dispositivos que lo incorporen podrán operar a una alta velocidad de datos. Hay varios de estos dispositivos disponibles comercialmente.

16.19 U.S. Department of Defense: *Trusted Computer System Evaluation Criteria* (el "Libro naranja"), documento No. DoD 5200-28-STD. DoD National Computer Security Center (diciembre, 1985).

16.20 U.S. National Computer Security Center: *Trusted Database Management System Interpretation of Trusted Computer System Evaluation Criteria* (el "Libro Lavanda"), documento No. NCSC-TG-201, Versión I (abril, 1991).

16.21 Marianne Winslett, Kenneth Smith y Xiaolei Qian: "Formal Query Languages for Secure Relational Databases", *ACM TODS 19*, No. 4 (diciembre, 1994).

Continúa el trabajo de la referencia [16.16].

16.22 Ron Wolf: "How Safe Is Computer Data? A Lot of Factors Govern the Answer", *San Jose Mercury News* (julio 5, 1990).

RESPUESTAS A EJERCICIOS SELECCIONADOS

16.1

a.
```
AUTHORITY AM
    GRANT RETRIEVE ON STATS TO Ford ;
```

b.
```
AUTHORITY BBB
    GRANT INSERT, DELETE ON STATS TO Smith ;
```

C.
```
AUTHORITY CCC
    GRANT RETRIEVE
    ON   STATS
    WHEN USER () ■ NOMBRE
    TO   ALL ;
```

Estamos suponiendo que los usuarios usan su propio nombre como ID de usuario. Observe el uso de una cláusula WHEN y el operador niládico integrado USER().

d.
```
AUTHORITY DDD
    GRANT RETRIEVE, UPDATE ( SALARIO, IMPUESTOS ) ON
    STATS TO   Nash ;
```

e.
```
AUTHORITY EEE
    GRANT RETRIEVE ( NOMBRE, SALARIO, IMPUESTOS )
    ON   STATS TO   Todd ;
```

f.
```
AUTHORITY FFF
    GRANT RETRIEVE ( NOMBRE, SALARIO, IMPUESTOS ),
          UPDATE ( SALARIO, IMPUESTOS )
    ON   STATS TO   Ward ;
```

```
g. VAR PREDICADORES VIEW
       STATS WHERE OCUPACIÓN = 'Predicador' ;

   AUTHORITY GGG
       GRANT ALL
       ON    PREDICADORES
       TO    Pope ;
```

Observe la necesidad de usar una vista en este ejemplo.

```
h. VAR NOESPECIALISTA VIEW
       WITH ( STATS RENAME OCUPACIÓN AS X ) AS T1 ,
            ( EXTEND STATS
            ADD COUNT ( T1 WHERE X = OCUPACIÓN ) AS Y ) AS T2, ( T2
            WHERE Y > 10 ) AS T3 : T3 { ALL BUT Y }

   AUTHORITY HHH
       GRANT DELETE
       ON    NOESPECIALISTA
       TO   Jones ;

 i. VAR TRABAJOMAXMIN VIEW
   WITH ( STATS RENAME OCUPACIÓN AS X ) AS T1 ,
        ( EXTEND STATS
          ADD MAX ( T1 WHERE X = OCUPACIÓN, SALARIO ) AS SALMAX,
              MIN ( T1 WHERE X = OCUPACIÓN, SALARIO ) AS SALMIN )
              AS T2 :

   T2 { OCUPACIÓN, SALMAX, SALMIN }

   AUTHORITY III
       GRANT RETRIEVE ON TRABAJOMAXMIN TO King ;
```

16.2 Aquí sólo hacemos una observación: un usuario que tiene la autoridad para crear una nueva va rrel base (y de hecho lo hace) puede ser visto como el **propietario** de esa nueva varrel. Al igual que como sucede en SQL, al propietario de una varrel base dada se le deben otorgar automáticamente todos los privilegios posibles sobre esa varrel, incluyendo no solamente los privilegios de RE TRIEVE, INSERT, UPDATE y DELETE (por supuesto), sino también los privilegios para definir au toridades que otorguen privilegios a otros usuarios sobre esa varrel.

16.3 Un rastreador individual para Hal es

```
HIJOS > 1  AND NOT  (  OCUPACIÓN =   'Constructor de casas'   )
```

Considere la siguiente secuencia de consultas:

```
COUNT ( STATS WHERE HIJOS > 1 )
```

Resultado: 6.

```
COUNT ( STATS WHERE HIJOS > 1 AND NOT
                  ( OCUPACIÓN =  'Constructor de casas'  )  )
```

Resultado: 5.

Por lo tanto, la expresión

```
HIJOS > 1 AND OCUPACIÓN ■ 'Constructor de casas'
```

Identifica en forma única a Hal.

»

```
SUM ( STATS WHERE HIJOS > 1, IMPUESTOS )
```

Resultado: 48 K.

```
SUM ( STATS WHERE HIJOS > 1 AND NOT
                ( OCUPACIÓN = 'Constructor de casas' ), IMPUESTOS )
```

Resultado: 46K.

Por lo tanto, la cifra de impuestos de Hal es 2K.

16.4 Rastreador general: SEXO = 'P.

```
SUM ( STATS WHERE SEXO = 'F1, IMPUESTOS )
```

Resultado: 70 K.

```
SUM ( STATS WHERE NOT ( SEXO • 'F' ), IMPUESTOS )
```

Resultado: 16K.

Por lo tanto, el impuesto total es 86K.

```
SUM ( STATS WHERE ( HIJOS > 1 ANO
                    OCUPACIÓN = 'Constructor de casas1 ) OR
                  SEXO = ' F' , IMPUESTOS )
```

Resultado: 72K.

```
SUM ( STATS WHERE ( HIJOS > 1 AND
                    OCUPACIÓN = 'Constructor de casas1 ) OR NOT (
                  SEXO = ' F' ), IMPUESTOS )
```

Resultado: 16K.

Si sumamos estos resultados y restamos el total previamente calculado, tenemos la cifra de impuesto de Hal = 88K - 86K = 2K. |

16.5 El texto plano es

```
EYES I DARE NOT MEET IN DREAMS
```

¿Cuál es la clave de cifrado?

16.7 Un problema es que, aun en un sistema que soporta cifrado, los datos deben ser procesados internamente en forma de texto plano (para que las comparaciones operen correctamente) y por lo tanto sigue existiendo el riego de que los datos delicados sean accesibles desde aplicaciones ejecutadas en forma concurrente o que aparezcan en un vaciado de memoria. También existen algunos problemas técnicos para el indexado de datos cifrados y el mantenimiento de registros de bitácora para esos datos.

16.8

a.
```
GRANT SELECT ON STATS TO Ford ;
```

b.
```
GRANT INSERT, DELETE ON STATS TO Smith ;
```

C.
```
CREATE VIEW MIA AS
        SELECT STATS.*
        FROM  STATS
        WHERE  STATS.NOMBRE • CURRENTUSER ;

GRANT SELECT ON MIA TO PUBLIC ;
```

Aquí estamos suponiendo que los usuarios usan su propio nombre como ID de usuario. Observe el uso del operador niládico integrado CURRENT_USER.

```
d. GRANT SELECT, UPDATE ( SALARIO, IMPUESTOS ) ON STATS TO Nash ;

e. CREATE VIEW UST AS
          SELECT STATS.NOMBRE, STATS.SALAR10, STATS.IMPUESTOS
          FROM  STATS ;

   GRANT SELECT ON UST TO Todd ;
```

Aquí, SQL tiene que usar una vista debido a que no soporta privilegios de SELECT específicos de columna.

```
f. GRANT SELECT, UPDATE ( SALARIO, IMPUESTOS ) ON UST TO Ward ;
```

Esta solución usa la misma vista que la anterior.

```
g. CREATE VIEW PREDICADORES AS
          SELECT STATS.*
          FROM  STATS
          WHERE STATS.OCUPACIÓN = 'Predicador' ;

   GRANT ALL PRIVILEGES ON PREDICADORES TO Pope ¡
```

Observe el uso de la abreviatura "ALL PRIVILEGES" en este ejemplo. Sin embargo, ALL PRIVILEGES no significa literalmente todos los privilegios, sino sólo todos los privilegios sobre el objeto relevante para el cual el usuario que está emitiendo el GRANT tiene autoridad de otorgamiento.

```
h. CREATE VIEW NOESPECIALISTA AS SELECT STX.* FROM  STATS AS
          STX WHERE  ( SELECT COUNT (*) FROM STATS AS STY WHERE
          STY.OCUPACIÓN = STX.OCUPACIÓN ) > 10 ;

   GRANT DELETE ON NOESPECIALISTA TO Jones ;

i. CREATE VIEW TRABAJOMAXMIN AS
          SELECT STATS.OCUPACIÓN,
                 MAX ( STATS.SALARIO ) AS SALMAX,
                 MIN ( STATS.SALARIO ) AS SALMIN
          FROM  STATS GROUP BY STATS.OCUPACIÓN ;

   GRANT SELECT ON TRABAJOMAXMIN TO King ;
```

16.9

```
a. REVOKE SELECT ON STATS FROM Ford RESTRICT ;

b. REVOKE INSERT, DELETE ON STATS FROM Smith RESTRICT ;

C. REVOKE SELECT ON MIA FROM PUBLIC RESTRICT ;

d. REVOKE SELECT, UPDATE ( SALARIO, IMPUESTOS )
                          ON STATS FROM Nash RESTRICT ;
```

```
e.  REVOKE SELECT ON UST FROM Todd RESTRICT ;

f.  REVOKE SELECT, UPDATE ( SALARIO, IMPUESTOS )
                              ON UST FROM Ward RESTRICT ;

g.  REVOKE ALL PRIVILEGES ON PREDICADORES FROM Pope RESTRICT ;

h. REVOKE DELETE ON NOESPECIALISTA FROM Jones RESTRICT ;

 i. REVOKE SELECT ON TRABAJOMAXMIN FROM King RESTRICT ;
```

CAPITULO 17

Optimización

17.1 INTRODUCCIÓN

La optimización representa tanto un reto como una oportunidad para los sistemas relaciónales, un reto ya que si queremos que un sistema de esta naturaleza tenga un desempeño aceptable, será necesario optimizarlo; y una oportunidad puesto que una de las ventajas del enfoque relacional es precisamente que las expresiones relaciónales están en un nivel semántico lo suficientemente alto para que la optimización sea factible. Por el contrario, en un sistema no relacional —donde las peticiones de los usuarios están expresadas en un nivel semántico más bajo— cualquier "optimización" tiene que ser realizada en forma manual por el usuario ("optimización" entre comillas, ya que el término se utiliza generalmente para dar a entender optimización *automática).* En dichos sistemas es el usuario, y no el sistema, quien decide qué operaciones de bajo nivel se necesitan y en qué secuencia deben ser ejecutadas; por lo que si el usuario toma una mala decisión, no hay nada que el sistema pueda hacer para mejorar las cosas. Observe también lo que implica el hecho de que, en sistemas de este tipo, sea necesaria cierta experiencia en programación por parte del usuario; este hecho pone al sistema fuera del alcance de muchos usuarios que, de no ser así, podrían beneficiarse.

La ventaja de la optimización automática no es únicamente que los usuarios no tienen que preocuparse por formular sus consultas de la mejor manera (es decir, por cómo formular las peticiones para obtener el mejor desempeño del sistema). El hecho es que existe una posibilidad real de que el optimizador pueda hacerlo *mejor* que un usuario humano. Existen varias razones para esta situación, entre las que se encuentran las siguientes:

1. Un buen optimizador —¡y tal vez debemos enfatizar "buen"!— tendrá una gran cantidad de información disponible que por lo general no tienen los usuarios humanos; en particular conocerá determinada información **estadística** tal como:

   - La cantidad de valores en cada dominio.
   - La cantidad actual de tupias en cada varrel base.
   - La cantidad actual de valores distintos en cada atributo de cada varrel base.
   - La cantidad de veces que tales valores se dan en cada uno de esos atributos.

   y así sucesivamente. (Toda esta información se mantendrá en el catálogo del sistema; vea la sección 17.5.) Por consecuencia, el optimizador deberá ser capaz de hacer una valoración más precisa de la eficiencia de cualquier estrategia dada para implementar una petición en particular, y por lo tanto será más probable que escoja la implementación más eficiente.

2. Además, si las estadísticas de la base de datos cambian con el tiempo, tal vez sea necesaria una selección de estrategia diferente; en otras palabras, es posible que se requiera una *re-optimización.* En un sistema relational, la reoptimización es trivial, ya que simplemente in volucra un reprocesamiento de la petición relacional original a cargo del optimizador del sistema. Por el contrario, en un sistema que no es relacional, la reoptimización involucra la reescritura del programa y es muy probable que no se realice nunca.
3. Por otra parte, el optimizador es un *programa* y es por definición mucho más paciente que un usuario humano típico. El optimizador es bastante capaz de considerar literalmente cien tos de estrategias de Ímplementación diferentes para una petición dada, y en cambio es muy poco probable que un usuario humano llegue a considerar más de tres o cuatro (al menos a profundidad).
4. Por último, el optimizador puede ser considerado, en cierto sentido, como la personificación de las habilidades y servicios de "los mejores" programadores humanos. Como consecuen cia, tiene el efecto de poner a disposición de *todos* esas habilidades y servicios, lo que signi fica —por supuesto— que está poniendo a disposición de un amplio rango de usuarios, un conjunto de recursos en una forma eficiente y económica.

Todo lo anterior debe servir como evidencia para apoyar la afirmación hecha al inicio de esta sección: la **capacidad de optimization** —es decir, el hecho de que las peticiones relacionales sean optimizables— es de hecho una *ventaja* de los sistemas relaciónales.

Entonces, el propósito general del optimizador es seleccionar una estrategia eficiente para la evaluación de una expresión relacional dada. En este capítulo describiremos algunos de los principios y técnicas fundamentales involucrados en este proceso. Después de un ejemplo introductorio y motivador en la sección 17.2, la sección 17.3 presenta un panorama general de la manera en que funcionan los optimizadores. Por su parte, la sección 17.4 describe un aspecto muy importante del proceso: la *transformación de la expresión* (también conocida como *reescritura de la consulta).* La sección 17.5 explica brevemente la cuestión de las *estadísticas de la base de datos.* Posteriormente, la sección 17.6 describe con cierto detalle un enfoque específico de la optimización, llamado *descomposición de la consulta.* Después, la sección 17.7 trata el asunto de cómo se implementan los operadores relaciónales (las juntas, etcétera) y trata brevemente el uso de las estadísticas que explicamos en la sección 17.5, para realizar una estimación de los costos. Por último, la sección 17.8 presenta un resumen de todo el capítulo.

Un último comentario introductorio: es común referirnos específicamente a este tema como optimización de *consultas.* Sin embargo, este término es un poco engañoso, ya que la expresión a ser optimizada (es decir, la "consulta") pudo por supuesto haberse presentado en algún contexto diferente a la interrogación interactiva de la base de datos; en particular, puede ser parte de una operación de actualización en lugar de una consulta como tal. Lo que es más, el término *optimización* es en cierta forma una exageración, ya que por lo general no hay ninguna garantía de que la estrategia de Ímplementación elegida sea en realidad *óptima* en cualquier sentido mensurable; de hecho podría serlo, pero generalmente lo único que se sabe con certeza es que la estrategia "optimizada" es una *mejora* de la versión original no optimizada. (Sin embargo, en ciertos contextos bastante limitados podríamos afirmar legítimamente que la estrategia seleccionada es óptima en un sentido muy específico; vea por ejemplo la referencia [17.31]).

17.2 UN EJEMPLO MOTIVADOR

Comenzamos con un ejemplo sencillo, una ampliación de uno que ya explicamos brevemente en el capítulo 6, sección 6.6 y que da una idea de las mejoras dramáticas que son posibles. La consulta es "obtener el nombre de los proveedores que proporcionan la parte P2". Una formulación algebraica de esta consulta es:

```
( ( VP JOIN V ) WHERE P# = P# ( ' P2' ) ) { PROVEEDOR }
```

Supongamos que la base de datos contiene 100 proveedores y 10,000 envíos, de los cuales sólo 50 son de la parte P2. Por razones de simplicidad, supongamos que las varrels V y VP están representadas expresamente en el disco como dos archivos guardados por separado (con un registro guardado por tupia). Entonces, si el sistema tuviera simplemente que evaluar la expresión "directamente" —es decir, sin ninguna optimización— la secuencia de eventos sería la siguiente:

1. **Juntar VP y V (sobre V#).** Este paso involucra la lectura de 10,000 envíos, la lectura de cada uno de los 100 proveedores 10,000 veces (una vez para cada uno de los 10,000 envíos), la construcción de un resultado intermedio que conste de las 10,000 tupias juntadas y la escri tura de esas 10,000 tupias juntadas de nuevo en el disco (para efectos del ejemplo, suponemos que no hay espacio en la memoria principal para este resultado intermedio).
2. **Restringir el resultado del paso 1 sólo a las tupias para la parte P2.** Este paso involu cra la lectura de las 10,000 tupias juntadas nuevamente hacia memoria, pero produce un re sultado que consiste solamente en 50 tupias, el cual suponemos es lo suficientemente pequeño para conservarse en la memoria principal.
3. **Proyectar el resultado del paso 2 sobre PROVEEDOR.** Este paso produce el resultado final deseado (como máximo 50 tupias, las cuales pueden permanecer en memoria principal).

El siguiente procedimiento es equivalente al que acabamos de describir, en el sentido de que produce necesariamente el mismo resultado final (aunque es claramente mucho más eficiente).

1. **Restringir VP solamente a las tupias para la parte P2.** Este paso involucra la lectura de 10,000 tupias, pero produce un resultado que consiste solamente en 50 tupias, las cuales suponemos que se mantendrán en memoria principal.
2. **Juntar el resultado del paso 1 con V (sobre V#).** Este paso involucra la lectura de 100 pro veedores (por supuesto solamente una vez y no una por cada envío de P2) y produce nueva mente un resultado de 50 tupias (que siguen en memoria principal).
3. **Proyectar el resultado del paso 2 sobre PROVEEDOR** (igual que en el paso 3 anterior). El resultado final deseado (como máximo 50 tupias) permanece en memoria principal.

El primero de estos dos procedimientos involucra la E/S de 1,030,000 tupias mientras que el segundo involucra solamente 10,100. Por lo tanto, queda claro que si tomamos a la "cantidad de E/S de las tupias" como medida de desempeño, el segundo procedimiento es un poco más de 100 veces mejor que el primero. (Por supuesto, en la práctica lo que importa es la E/S de las *páginas* y no la de tupias, aunque para los propósitos actuales podemos ignorar esta precisión.) ¡También queda claro que preferiríamos que la implementación use el segundo procedimiento en lugar del primero!

Por lo tanto, vemos que un cambio muy simple en el algoritmo de ejecución (hacer una restricción y luego una junta, en lugar de una junta y luego una restricción) ha producido una mejora drástica en el desempeño. Y la mejora sería todavía más drástica si los envíos estuvieran además **indexados** o **dispersados** en P#, ya que la cantidad de tupias de envíos leídas en el paso 1 se reduciría de 10,000 a sólo 50, y el nuevo procedimiento sería entonces casi 7,000 veces mejor que el original. En forma similar, si los proveedores también estuvieran indexados o dispersados en V#, la cantidad de tupias de proveedores leídas en el paso 2 se reduciría de 100 a 50, y así el procedimiento sería ahora más de 10,000 veces mejor que el original. Esto significa que si la consulta original no optimizada tomara tres horas para ejecutarse, la versión final sería ejecutada en un tiempo menor a *un segundo.* Y por supuesto, todavía son posibles otras mejoras adicionales.

El ejemplo anterior aunque sencillo, debería ser suficiente para dar una idea de por qué es necesaria la optimización. También debería dar alguna idea de los tipos de mejoras posibles en la práctica. En la siguiente sección presentaremos un panorama general de un enfoque sistemático al problema de la optimización; en particular mostraremos cómo podemos dividir el problema general en una serie de subproblemas más o menos independientes. Este panorama proporciona un marco de trabajo conveniente en el cual es posible explicar y comprender las estrategias y técnicas de optimización individuales como las que explico en secciones posteriores.

17.3 UN PANORAMA GENERAL DEL PROCESAMIENTO DE CONSULTAS

Podemos identificar cuatro grandes etapas en el procesamiento de consultas, de la siguiente forma (consulte la figura 17.1):

1. Convertir la consulta a su forma interna.
2. Convertirla a la forma canónica.
3. Seleccionar procedimientos candidatos de bajo nivel.
4. Generar planes de consulta y seleccionar el más barato.

Ahora procederemos a ampliar cada una de estas etapas.

Etapa 1: Convertir la consulta a su forma interna

La primera etapa involucra la conversión de la consulta original en alguna representación interna que sea más adecuada para manejarla en la máquina, eliminando así consideraciones meramente externas (tales como los detalles de la sintaxis concreta del lenguaje de consulta que se está considerando) y allanando el camino para las etapas subsecuentes del proceso de optimización. *Nota:* El procesamiento de vistas (es decir, el proceso de reemplazar las referencias a las vistas por las expresiones de definición de vistas aplicables) también se realiza durante esta etapa.

La pregunta obvia es: ¿en qué formalismo deberá estar basada la representación interna? Por supuesto, sin importar cuál formalismo se elija, éste deberá ser lo suficientemente amplio para representar todas las consultas posibles en el lenguaje de consulta externo. También deberá ser lo más neutral posible, en el sentido de que no deberá interferir con las selecciones subsecuentes. Por lo general, la forma interna seleccionada es algún tipo de **árbol de sintaxis abstracto** o **árbol de consulta.** Por ejemplo, la figura 17.2 muestra una posible representación de árbol de

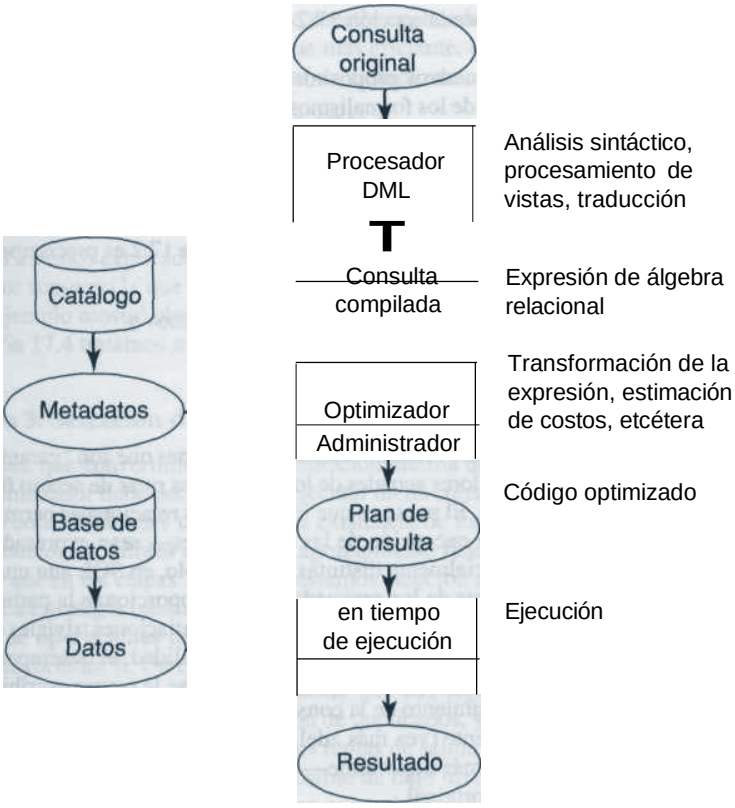


Figura 17.1 Panorama general del procesamiento de consultas.

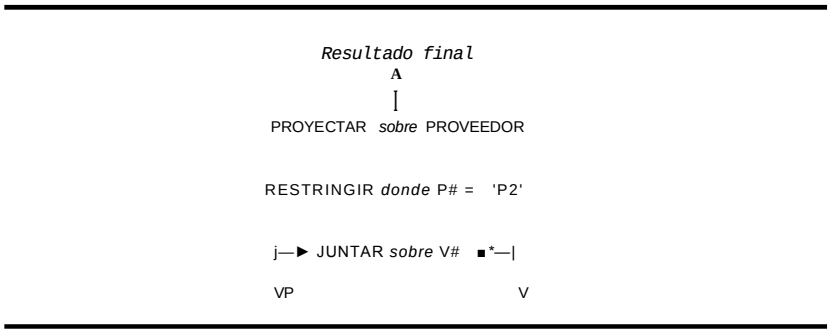


Figura 17.2 "Obtener los nombres de los proveedores que proporcionan la parte P2" (árbol de consulta).

consulta para el ejemplo de la sección 17.2 ("obtener los nombres de los proveedores que proporcionan la parte P2").

Sin embargo, para nuestros propósitos es más conveniente suponer que la representación interna representa alguno de los formalismos con los que ya estamos familiarizados; en concreto, el álgebra relacional o el cálculo relacional. Un árbol de consulta como el de la figura 17.2 puede ser considerado simplemente como una representación codificada alternativa para alguna expresión de uno de esos dos formalismos. Para ordenar nuestras ideas, aquí suponemos que el formalismo se refiere específicamente al álgebra. Por lo tanto, a partir de ahora daremos por hecho que la representación interna de la consulta de la figura 17.2 es precisamente la expresión algebraica que mostré anteriormente:

```
( ( VP JOIN V ) WHERE P# = P# ( 'P2' ) ) { PROVEEDOR }
```

Etapa 2: Conversión a la forma canónica

En esta etapa, el optimizador realiza varias optimizaciones que son "garantizadas como buenas", sin tomar en cuenta los valores actuales de los datos ni las rutas de acceso físicas que existen en la base de datos almacenada. El punto es que los lenguajes relaciónales permiten generalmente que todas las consultas —con excepción de las más simples— sean expresadas en diversas formas que son al menos superficialmente distintas. Por ejemplo, en SQL aun una consulta tan sencilla como "obtener los nombres de los proveedores que proporcionan la parte P2" puede ser expresada en docenas de formas diferentes,* sin contar las variaciones triviales como el reemplazo de *A = B* por *B = A* o de *p* AND *q* por *q* AND *p*. Y en realidad, el desempeño de una consulta no debe depender de la forma particular en que al usuario se le ocurra escribirla. Por lo tanto, el siguiente paso en el procesamiento de la consulta es convertir la representación interna en alguna **forma canónica** equivalente (vea más adelante), con el objeto de eliminar esas diferencias superficiales y —lo que es más importante— encontrar una representación que sea, en cierta forma, más eficiente que la original.

Una nota con relación a la "forma canónica": La noción de *forma canónica* es fundamental en muchas ramas de las matemáticas y disciplinas relacionadas. Podemos definirla de la siguiente manera: Dado un conjunto de objetos *Q* (digamos consultas) y una noción de equivalencia entre esos objetos (digamos la noción de que las consultas *ql* y *q2* son equivalentes si y sólo si producen necesariamente el mismo resultado), decimos que el subconjunto C de *Q* es un **conjunto de formas canónicas** para *Q* bajo la definición establecida de equivalencia, si y sólo si cada objeto *q* en *Q* es equivalente a sólo un objeto *c* en *C*. Decimos que el objeto c es *Informa canónica* para el objeto *q*. Todas las propiedades "interesantes" aplicadas al objeto *q* también son aplicadas a su forma canónica c; por lo tanto, es suficiente estudiar sólo el pequeño conjunto C, y no el conjunto grande *Q*, para probar una variedad de resultados "interesantes".

---

*Sin embargo, debemos señalar que el lenguaje SQL es excepcionalmente propenso a este problema (vea el ejercicio 7.12 en el capítulo 7 y también la referencia [4.18]). Por lo general, otros lenguajes (por ejemplo, el álgebra o el cálculo) no proporcionan tantas formas diferentes para hacer las mismas cosas. De hecho, esta "flexibilidad" innecesaria por parte de SQL hace la vida más difícil para el *implementador* (por no mencionar al usuario), ya que hace el trabajo del optimizador más difícil.

Regresemos al punto central de nuestra explicación. Para transformar la salida de la etapa 1 en alguna forma equivalente, aunque más eficiente, el optimizador utiliza determinadas **reglas** o **leyes de transformación.** He aquí un ejemplo de estas reglas: la expresión

```
( A JOIN B ) WHERE restricción sobre A
```

puede ser transformada a la expresión equivalente, pero más eficiente

```
( A WHERE restricción sobre A ) JOIN B
```

Ya hemos explicado brevemente esta transformación en el capítulo 6, sección 6.6; de hecho, fue por supuesto la que estuvimos usando en nuestro ejemplo introductorio de la sección 17.2, y el ejemplo mostró claramente por qué es necesaria dicha transformación. Más adelante, en la sección 17.4 tratamos más reglas de transformación.

Etapa 3: Selección de procedimientos candidatos de bajo nivel

Una vez que convertimos la representación interna de la consulta en una forma más adecuada, el optimizador debe decidir cómo ejecutar dicha consulta transformada. En esta etapa entran en juego consideraciones tales como la existencia de los índices u otras rutas de acceso físicas, la distribución de valores de datos, el agrupamiento físico de los datos almacenados, etcétera. Observe que en las etapas 1 y 2, no prestamos atención a estos asuntos.

La estrategia básica es considerar a la expresión de consulta como la especificación de una serie de **operaciones de "bajo nivel"** (juntar, restringir, resumir, etcétera), con cierta interdependencia entre sí. Un ejemplo de tal interdependencia es el siguiente: para realizar una proyección, el código requerirá generalmente que sus tupias de entrada sean ordenadas en alguna secuencia que permita la eliminación de duplicados, y esto significa que la operación inmediata anterior de la serie debe producir sus tupias de salida en la misma secuencia.

Ahora, para cada operación posible de bajo nivel —y probablemente también para diversas combinaciones comunes de tales operaciones— el optimizador tendrá a su disposición un conjunto de **procedimientos de implementación** predefinidos. Por ejemplo, habrá un conjunto de procedimientos para la implementación de la operación de restricción: uno para el caso en donde la restricción es una prueba de igualdad sobre una clave candidata, uno donde el atributo de restricción está indexado, uno donde esté disperso, etcétera. La sección 17.7 muestra ejemplos de dichos procedimientos (vea también las referencias [17.8] a [17.14]).

Cada procedimiento tendrá también una **fórmula de costo** (con parámetros) asociada, que indica el costo de ejecutar ese procedimiento (generalmente en términos de E/S de disco, aunque algunos sistemas también toman en cuenta la utilización de la CPU y otros factores). Estas fórmulas de costo se usan en la etapa 4 (vea más adelante). Las referencias [17.8] a [17.14] analizan y explican las fórmulas de costo para algunos procedimientos de implementación diferentes, bajo una variedad de suposiciones distintas. Posteriormente, vea también la sección 17.7.

Por lo tanto, si utilizamos la información del catálogo referente al estado actual de la base de datos (existencia de índices, cardinalidades actuales, etcétera) y usamos también la información de interdependencia que mencioné anteriormente, el optimizador seleccionará uno o más procedimientos candidatos para la implementación de cada una de las operaciones de bajo nivel en la expresión de consulta. En ocasiones, a este proceso se le llama **selección de ruta de acceso** (vea las referencias [17.34] y [17.35]). *Nota:* De hecho, las referencias [17.34] y [17.35]

usan el término *selección de ruta de acceso* para cubrir las etapas 3 y 4 (no sólo la 3). Además, en la práctica puede ser difícil separar claramente las dos, ya que la etapa 3 fluye en forma más o menos suave hacia la etapa 4.

Etapa 4: Generación de los planes de consulta y selección del más barato

La última etapa del proceso de optimización involucra la construcción de un conjunto de **planes de consulta** candidatos, seguida de una selección del mejor de esos planes (es decir, el más barato). Cada plan de consulta es construido por medio de la combinación de una serie de procedimientos de implementación candidatos; uno de estos procedimientos para cada una de las operaciones de bajo nivel en la consulta. Observe que generalmente existirán muchos planes posibles (tal vez demasiados) para una consulta dada. De hecho, en la práctica tal vez no sea buena idea generar todos los planes posibles, ya que en combinación habrá muchos y la tarea de seleccionar al más barato bien puede llegar a ser excesivamente cara por sí misma; por lo tanto, es muy necesaria —aunque no esencial— alguna técnica para mantener dentro de límites razonables al conjunto generado (pero vea la referencia [17.55]). Por lo general, al hecho de "mantener el conjunto dentro de límites" se le llama *reducción del espacio de búsqueda*, ya que puede ser considerado como la reducción —a proporciones manejables— del rango ("espacio") de posibilidades que el optimizador debe examinar ("buscar").

Naturalmente, la selección del plan más barato requiere de un método para asignar un costo a cualquier plan dado. Por supuesto, el costo de un plan dado es básicamente la simple suma de los costos de los procedimientos individuales que conforman ese plan y por lo tanto, lo que el optimizador tiene que hacer es evaluar las fórmulas de costo de esos procedimientos individuales. El problema es que esas fórmulas de costo dependerán del tamaño de las relaciones a procesar, y debido a que casi todas —con excepción de las consultas más sencillas— involucran la generación de resultados intermedios durante la ejecución, el optimizador tendrá que estimar el tamaño de esos resultados intermedios para evaluar las fórmulas. Por desgracia, esos tamaños tienden a ser muy dependientes de los valores de datos actuales. En consecuencia, la estimación precisa de los costos puede ser un problema difícil. Las referencias [17.3] y [17.4] explican algunos enfoques a este problema y dan referencias hacia otras investigaciones en el área.

17.4 TRANSFORMACIÓN DE EXPRESIONES

En esta sección describimos algunas reglas de transformación que pueden ser útiles en la etapa 2 del proceso de optimización. Dejamos como ejercicio algunos ejemplos para ilustrar las reglas y decidir exactamente por qué pueden ser útiles.

Por supuesto, deberá entender que dada una expresión específica a transformar, la aplicación de una regla podría generar una expresión que luego podrá ser transformada de acuerdo con alguna otra regla. Por ejemplo, es poco probable que la consulta original haya sido expresada directamente en forma tal que requiera dos proyecciones sucesivas —vea la segunda regla en la subsección "Restricciones y proyecciones", que viene a continuación—; aunque tal expresión puede ser presentada internamente como resultado de la aplicación de otras transformaciones. (Un caso importante lo proporciona el **procesamiento de vistas.** Por ejemplo, considere la consulta "obtener todas las ciudades de la vista VC", donde la vista VC está definida como la proyección de los proveedores sobre V# y CIUDAD.) En otras palabras, a partir de la expresión original, el

optimizador aplicará repetidamente sus reglas de transformación hasta que finalmente llegue a una expresión que juzgue "óptima" para la consulta en consideración (de acuerdo con algún conjunto integrado de técnicas).

Restricciones y proyecciones

Aquí aparecen primero algunas transformaciones que involucran solamente restricciones y proyecciones.

1. Una secuencia de restricciones sobre la misma relación puede ser transformada en una sola restricción (con AND) sobre esa relación. Es decir, la expresión

   (*A* WHERE *restricciónі*) WHERE *restricción2*

   es equivalente a la expresión

   A WHERE *restriccióm* AND *restríccióm*

2. En una secuencia de proyecciones frente a la misma relación, es posible ignorar todas con excepción de la última. Es decir, la expresión

   (*A* { *atributosi* }) { *atributos2* } es

   equivalente a la expresión

   A { *atributos2* }

   Por supuesto, cada uno de los atributos mencionados en *atributos2,* también debe ser mencionado en *atributosi* para que la expresión original tenga sentido.

3. Una restricción de una proyección puede ser transformada en una proyección de una res tricción. Es decir, la expresión

   (*A* { *atributos* }) WHERE *restricción*

   es equivalente a la expresión

   (*A* WHERE *restricción*) { *atributos* }

   Por lo general, observe que es una buena idea hacer las restricciones antes que las proyecciones, ya que el efecto de la restricción será reducir el tamaño de la entrada a la proyección y por lo tanto, reducir la cantidad de datos que sena necesario ordenar para efectos de eliminación de duplicados.

Distributividad

De hecho, la regla de transformación utilizada en el ejemplo de la sección 17.2 (la transformación de una junta seguida por una restricción, en una restricción seguida por una junta) es un caso especial de una ley más general llamada ley *distributiva.* En general, decimos que el operador monádico/está **distribuido** sobre el operador diádico O si y sólo si

f (A o B) = f (A) o f (B)

para toda A y *B*. Por ejemplo, en la aritmética ordinaria, SQRT (raíz cuadrada) se distribuye sobre la multiplicación, porque

```
SQRT ( A * B ) = SQRT ( A ) * SQRT ( B )
```

para toda A y *B*. Por lo tanto, al transformar expresiones aritméticas, un optimizador de expresiones aritméticas siempre puede reemplazar a cualquiera de estas expresiones por la otra. Como ejemplo contrario, SQRT no se distribuye sobre la suma, ya que la raíz cuadrada de A + *B* no es igual a la suma de las raíces cuadradas de A y *B*, en general.

En el álgebra relacional, el operador de **restricción** se distribuye sobre la **unión,** la **intersección** y la **diferencia.** También se distribuye sobre la **junta** si y sólo si la condición de restricción consiste —en su forma más compleja— en dos condiciones de restricción independientes unidas por AND (una para cada uno de los dos operandos de la junta). En el caso del ejemplo de la sección 17.2, este requerimiento es satisfecho, ya que la condición es de hecho muy simple y se aplica solamente a uno de los operandos (y por lo tanto, pudimos usar la ley distributiva para reemplazar la expresión por una equivalente más eficiente). El efecto neto fue que fuimos capaces de "aplicar tempranamente la restricción". La aplicación temprana de las restricciones casi siempre es una buena idea, ya que sirve para reducir la cantidad de tupias a revisar en la siguiente operación de la secuencia, y probablemente también reduce la cantidad de tupias en la salida de esa siguiente operación.

Estos son algunos casos más específicos de la ley distributiva, pero esta vez involucran la **proyección.** Primero, el operador de proyección se distribuye sobre la **unión** y la **intersección** (pero no sobre la **diferencia):**

```
( A UNION 6) { C } = A { C } UNION B { C }
( A INTERSECT S) { C } = A { C } INTERSECT 6 { C }
```

Por supuesto, A y *B* deben ser del mismo tipo. Segundo, la proyección también se distribuye sobre la **junta,** es decir,

```
( A JOIN B ) { C } = ( A { A C } ) JOIN ( 6 { BC > )
```

—rsi y sólo si:

- *AC* es la unión de (a) los atributos comunes a A y *B*, y (b) aquellos atributos de C que aparecen solamente en A; y
- *BC* es la unión de (a) los atributos comunes a A y *B*, y (b) aquellos atributos de C que aparecen solamente en *B*.

En general, estas leyes pueden ser usadas para "hacer proyecciones tempranas", lo cual nuevamente es buena idea por razones similares a las que dimos anteriormente para las restricciones.

Conmutatividad y asociatividad

Otras dos leyes generales importantes son las de *conmutatividad* y *asociatividad.* Primero, decimos que el operador diádico O es **conmutativo** si y sólo si

```
A o B = B o A
```

para toda A y *B.* Por ejemplo, en la aritmética, la multiplicación y la suma son conmutativas, pero la división y la resta no. En el álgebra relacional, la **unión,** la **intersección** y la **junta** son conmutativas, pero la **diferencia** y la **división** no lo son. Por lo tanto, si una consulta involucra (por ejemplo) una junta de dos relaciones *A* y *B,* la ley conmutativa significa que no hay diferencia lógica en que AoBse tome como la relación "externa" o la "interna". Por lo tanto, el sistema es libre de seleccionar (digamos) la relación más pequeña como la "externa" para calcular la junta (vea la sección 17.7).

Consideremos ahora la asociatividad. Decimos que el operador diádico O es **asociativo** si y sólo si

para toda A, fiyC.En la aritmética, la multiplicación y la suma son asociativas, pero la división y la resta no. En el álgebra relacional, la **unión,** la **intersección y** la **junta** son asociativas, pero la **diferencia** y la **división** no. Por lo tanto, si una consulta involucra (por ejemplo) a una junta de tres relaciones A, *B* y C, las leyes asociativa y conmutativa juntas significan que no hay diferencia lógica con respecto al orden en el cual se juntan las relaciones. Por lo tanto, el sistema es libre para decidir cuál de las diversas secuencias posibles es más eficiente.

Idempotencia

Otra ley general importante es la de *idempotencia.* Decimos que el operador diádico o es **idempotente** si y sólo si

```
A O A m A
```

para toda A. Como podríamos esperar, la propiedad de idempotencia también puede ser útil en la transformación de expresiones. En el álgebra relacional, la **unión,** la **intersección** y la **junta** son idempotentes, pero la **diferencia** y la **división** no lo son.

Expresiones computacionales escalares

No sólo las expresiones relaciónales están sujetas a las leyes de transformación. Por ejemplo, ya hemos indicado que determinadas transformaciones son válidas para las expresiones *aritméticas.* Éste es un ejemplo específico: la expresión

```
A * B + A * C
```

Puede ser transformada en

```
A * ( S + C )
```

en virtud del hecho que "*" se distribuye sobre "+". Un optimizador relacional necesita saber acerca de dichas transformaciones, ya que encontrará estas expresiones en el contexto de las operaciones de **extender y resumir.**

A propósito, observe que este ejemplo ilustra una forma ligeramente más general de la distributividad. Anteriormente, definimos la distributividad en términos de un operador *monádico* que se distribuye sobre un operador *diádico,* pero en este caso "*" y "+" son operadores

diádicos. En general, decimos que el operador diádico 5 se **distribuye** sobre el operador diádico O si y sólo si

A S (B o C) - (A S B) o' (A'S C) para toda A, *B* y *C* (en el ejemplo aritmético

anterior tome a 8 como "*" y a O como **"+")■**

Expresiones lógicas

Consideremos ahora a las expresiones *lógicas* (o expresiones de *verdad* o *"bolearías"* o *condicionales).* Supongamos que A y *B* son atributos de dos relaciones distintas. Entonces la expresión lógica

A > *B* AND e > 3

es claramente equivalente a lo siguiente (y por lo tanto, puede ser transformada en):

A > *B* AND 8 > 3 AND *A* > 3

La equivalencia se basa en el hecho de que el operador de comparación ">" es **transitivo.** Observe que esta transformación en verdad vale la pena, ya que permite que el sistema realice una restricción adicional (sobre A) antes de hacer la junta "mayor que" requerida por la comparación "A > *B".* Recuerde un punto anterior: por lo general, hacer las restricciones en forma temprana es buena idea; hacer que el sistema **infiera** restricciones "tempranas" adicionales, como en este caso, también es buena idea. *Nota:* Esta técnica está implementada en varios productos comerciales que incluyen, por ejemplo, a DB2 (donde se le llama "cierre transitivo de predicado") y a Ingres.

Éste es otro ejemplo: la expresión

A > *B* OR (*C* = *D* AND *E* < *F*)

puede ser transformada en

(*A* > *B* OR C = D) AND (*A* > *B* OR *E* < *F)*

en virtud del hecho de que OR se distribuye sobre AND. Este ejemplo ilustra otra ley general: cualquier expresión lógica puede ser transformada en una expresión equivalente en lo que se llama **forma normal conjuntiva** (CNF). Una expresión CNF es de la forma

C1 AND *C2* AND . . . AND *Cn*

donde cada uno de los *Cl, C2,..., Cn* es a su vez una expresión lógica (llamada un disyunción) que no involucra a ningún AND. La ventaja de la CNF es que una expresión CNF es *verdadera* sólo si cada una de las disyunciones es *verdadera y* —de forma similar— es *falsa* si cualquier disyunción *es falsa.* Puesto que AND es conmutativo (A AND *B* es lo mismo que *B* AND *A),* el optimizador puede evaluar las disyunciones individuales en cualquier orden; en particular, puede hacerlo en orden de dificultad creciente (primero las más fáciles). Tan pronto como encuentra una que es *falsa* puede detener todo el proceso. Además, en un sistema de procesamiento en paralelo podría incluso evaluar todas las disyunciones en paralelo [17.58] a [17.61]. Nuevamente, tan pronto como una disyunción produce *falso* es posible detener todo el proceso.

Deducimos de esta subsección y de la anterior, que el optimizador necesita saber cómo se aplican las propiedades generales (como la distributividad) no sólo a los operadores **relaciónales** (como la junta), sino también a los operadores de **comparación** (como ">"), los operadores **lógicos** (como AND y OR), los operadores **aritméticos** (como "+"), etcétera.

Transformaciones semánticas

Considere la siguiente expresión:

```
( VP JOIN V ) { P# }
```

Aquí, la junta es una *junta de clave externa con la clave candidata correspondiente;* hace coincidir una clave externa en VP con la clave candidata correspondiente de V. De esto se desprende que cada una de las tupias de VP se junta con alguna tupia de V y por lo tanto, cada una de las tupias de VP contribuye al resultado general con un valor P#. En otras palabras, ¡no hay necesidad de hacer la junta!, ya que la expresión puede simplificarse simplemente como

```
VP { P# }
```

Sin embargo, observe cuidadosamente que esta transformación es válida *sólo* debido a la semántica de la situación. Por lo general, cada uno de los operandos de una junta incluirá algunas tupias que no tienen contraparte en la otra (y por lo tanto, algunas tupias que no contribuyen al resultado general), por lo que las transformaciones como la que acabamos de ilustrar no son válidas. Sin embargo, en este caso cada una de las tupias de VP tiene una contraparte en V gracias a la restricción de integridad (de hecho, una restricción referencial) que dice que todo envío debe tener un proveedor y por lo tanto, a fin de cuentas la transformación es válida.

A una transformación que es válida sólo porque está en efecto una determinada restricción de integridad, se le llama una **transformación semántica** [17.27] y a la optimización resultante se le llama **optimización semántica.** La optimización semántica puede definirse como el proceso de transformar una consulta específica en otra cualitativamente diferente, pero que sin embargo garantiza producir el mismo resultado original gracias a que está garantizado que los datos satisfacen una determinada restricción de integridad.

Es importante comprender que en principio *cualquier restricción de integridad* puede ser usada en la optimización semántica (la técnica no está limitada a restricciones referenciales como en este ejemplo). Por ejemplo, suponga que la base de datos de proveedores y partes está sujeta a la restricción "todas las partes rojas deben estar almacenadas en Londres"; considere la consulta:

> Obtener los proveedores que proporcionan solamente partes rojas y se encuentran en la misma ciudad de al menos una de las partes que proveen.

¡Ésta es una consulta bastante compleja! Sin embargo, gracias a la restricción de integridad, podemos transformarla en la forma mucho más sencilla:

> Obtener los proveedores de Londres que solamente proporcionan partes rojas.

Nota: Hasta este momento, sé que sólo algunos pocos productos comerciales hacen algo por medio de la optimización semántica. Sin embargo, tal optimización podría en principio propor-

cionar mejoras de desempeño significativas, mejoras —muy probablemente— mayores a las que se obtienen por cualquiera de las técnicas tradicionales de optimización. Para una explicación adicional de la optimización semántica, vea las referencias [17.16], [17.28] a [17.30] y (en especial) la [17.27].

Conclusiones

Para cerrar esta sección, enfatizamos la importancia fundamental de la propiedad de **cierre** relational para todo lo que hemos explicado. El cierre significa que podemos escribir expresiones anidadas, lo que a su vez significa que una sola consulta puede ser representada por una sola expresión en vez de un procedimiento de varias expresiones y por lo tanto, no es necesario ningún análisis de flujo. Asimismo, esas expresiones anidadas están definidas en forma recursiva en términos de subexpresiones, las cuales permiten que el optimizador adopte una variedad de tácticas de evaluación del tipo "divide y vencerás" (vea la sección 17.6 más adelante). Y por supuesto, las diversas leyes generales —de distributividad, entre otras— ni siquiera comenzarían a tener sentido en ausencia del cierre.

17.5 ESTADÍSTICAS DE LA BASE DE DATOS

Las etapas 3 y 4 del proceso general de optimización —las etapas de "selección de la ruta de acceso"— utilizan las llamadas **estadísticas de la base de datos** que están guardadas en el catálogo (vea la sección 17.7 para obtener mayores detalles sobre la manera en que estas estadísticas son utilizadas). Para propósitos de ejemplificación resumiremos enseguida (con unos cuantos comentarios) algunas de las estadísticas principales que mantienen dos productos comerciales: DB2 e Ingres. En primer lugar, éstas son algunas de las estadísticas principales mantenidas por DB2:*

- Para cada *tabla base:*
  - Cardinalidad.
  - Número de páginas ocupadas por esta tabla.
  - Fracción del "espacio de tabla" ocupado por esta tabla.
- Para cada *columna* de cada tabla base:
  - Cantidad de valores distintos en esta columna.
  - Segundo valor más alto en esta columna.
  - Segundo valor más bajo en esta columna.
  - Sólo para columnas indexadas, los diez valores que suceden más frecuentemente en esta columna y la cantidad de veces que suceden.

---

*DB2 e Ingres usan los términos tabla y columna en vez de varrel y atributo, debido a que son sistemas SQL; por lo tanto así lo hacemos en esta sección. De hecho, observe que ambos productos también dan por hecho que las tablas base tienen correspondencia directa con las tablas *almacenadas.*

- Para cada *índice:*
  - Una indicación de si es un "índice de agrupamiento" (es decir, un índice que se usa para agrupar físicamente en el disco datos relacionados lógicamente).
  - De ser así, fracción de la tabla indexada que está todavía en secuencia de agrupamiento.
  - Número de páginas hoja en este índice.
  - Número de niveles en este índice.

Nota: Las estadísticas anteriores no son actualizadas cada vez que la base de datos se actualiza, debido a la sobrecarga que provocaría tal enfoque. En su lugar, son actualizadas en forma selectiva por medio de una utilería especial del sistema llamada RUNSTATS, la cual se ejecuta a petición del DBA (por ejemplo, después de una reorganización de la base de datos). Un comentario similar se aplica a la mayoría de los demás productos comerciales (aunque no a todos), incluyendo particularmente a Ingres (vea más adelante), donde la utilería se llama OPTIMIZEDB.

Éstas son algunas de las estadísticas principales de Ingres. *Nota:* En Ingres a un índice se le considera simplemente un caso especial de una tabla guardada y por lo tanto, las estadísticas que se muestran a continuación para las tablas y columnas base también pueden ser recolectadas para los índices.

- Para cada *tabla base:*
  - Cardinalidad.
  - Cantidad de páginas primarias para esta tabla.
  - Cantidad de páginas de desborde para esta tabla.
- Para cada *columna* de cada tabla base:
  - Cantidad de valores distintos en esta columna.
  - Valor máximo, mínimo y promedio para esta columna.
  - Valores actuales de esta columna y cantidad de veces que suceden.

17.6 UNA ESTRATEGIA DE DIVIDE Y VENCERÁS

Como mencioné al final de la sección 17.4, las expresiones relaciónales están definidas en forma recursiva en términos de subexpresiones y este hecho permite que el optimizador adopte una variedad de estrategias de "divide y vencerás". Observe que quizá tales estrategias son especialmente atractivas en un ambiente de procesamiento paralelo —en particular en un sistema distribuido— donde es posible ejecutar en paralelo, en procesadores diferentes, distintas partes de la consulta (vea las referencias [17.58] a [17.61]). En esta sección examinamos una de estas estrategias llamada **descomposición de la consulta,** la cual fue desarrollada por primera vez en el prototipo de Ingres (vea [17.36], [17.37]). *Nota:* Puede encontrar información adicional sobre la optimización en Ingres (más específicamente en el producto comercial, que en este aspecto es algo diferente al prototipo) en un artículo de Kooi y Frankforth que aparece en la referencia [17.2]. Vea también la referencia [17.38].

La idea básica detrás de la descomposición de la consulta, es dividir una consulta que involucra muchas variables de alcance* en una secuencia de consultas más pequeñas que generalmente involucran (cada una) a una o dos de estas variables, usando *separación* y *sustitución de tupias* para lograr la descomposición deseada:

- **Separación** es el proceso de eliminación de un componente de la consulta que tiene sólo una variable en común con el resto de la consulta.
- **Sustitución de tupia** es el proceso de sustitución de una variable de la consulta, una tupia a la vez.

Mientras sea posible, la separación siempre se aplica preferentemente en lugar de la sustitución de tupias (vea el ejemplo más adelante). Sin embargo, en ocasiones la consulta habrá sido descompuesta por medio de la separación en un conjunto de componentes que ya no pueden ser descompuestos usando esa técnica, y entonces entrará en juego la sustitución de tupias.

Damos un solo ejemplo (con base en el ejemplo de la referencia [17.36]). La consulta es "obtener los nombres de los proveedores de Londres que proporcionan alguna parte roja con un peso menor a 25 libras y en cantidad mayor a 200". Esta es una formulación en QUEL de esta consulta ("consulta Q0"):

```
: RETRIEVE ( V.PROVEEDOR )      V.CIUDAD =
                         AND    V.V#     = VP.V#
                         AND    VP. CANT > 200
                         AND    VP.P#    = P.P#
                         AND    P.COLOR  = "Rojo"
                         AND    P.PESO   < 25
```

Las variables de alcance (implícitas) son aquí V, P y VP, cada una con un rango sobre la varrel base del mismo nombre.

Ahora, si examinamos esta consulta podemos ver inmediatamente —a partir de los dos últimos términos de la comparación— que las únicas partes en las que estamos interesados son las que son rojas y pesan menos de 25 libras. Por lo tanto, podemos separar la "consulta de una variable" (de hecho, una proyección de una restricción) que involucra a la variable P:

```
D1 : RETRIEVE INTO P1 ( P.P# ) WHERE P.COLOR = "Rojo"
                                 AND  P.PESO < 25
```

Esta consulta de una variable es separable, ya que tiene solamente una variable en común (por decir algo, la propia P) con el resto de la consulta. Puesto que está vinculada con el resto de la consulta original por medio del atributo P# (en el término de comparación VP.P# = P.P#), el atributo P# es lo que debe aparecer dentro de la "prototupla" de la versión separada (vea el capítulo 7); es decir, la consulta separada debe recuperar exactamente los números de parte de las partes rojas que pesan menos de 25 libras. Conservamos esa consulta separada como la consulta DI que guarda su resultado en una varrel temporal P' (el efecto de la cláusula INTO es hacer que una nueva varrel P' —con el único atributo P#— se defina automáticamente para guardar el resultado de la ejecución del RETRIEVE). Por último, reemplazamos las referencias a P en la versión reducida de Q0 por referencias a P'. Refirámonos a esta nueva versión reducida como la consulta Ql:

```
Q1 : RETRIEVE ( V.PROVEEDOR ) WHERE V.CIUDAD = "Londres"
                               AND  V.V#    = VP.V#
                               AND  VP.CANT > 200
                               AND  VP.P#   = P1.P#
```

*Recuerde que el lenguaje de consulta de Ingres, QUEL, está basado en el cálculo.

Ahora realizamos un proceso similar de separación sobre la consulta Ql —separando la consulta de una variable que involucra a la variable VP como consulta D2— y dejamos una versión modificada de Ql (consulta Q2):

```
D2 : RETRIEVE INTO VP1 ( VP.V#, VP.P# ) WHERE VP.CANT > 200

Q2 : RETRIEVE ( V.PROVEEDOR ) WHERE V.CIUDAD = "Londres"
                              AND  V.V#    = VP1.V#
                              AND  VP'.P#  = P'.P#
```

Luego separamos la consulta de una variable que involucra a V:

```
D3 : RETRIEVE INTO V ( V.V#, V.PROVEEDOR ) WHERE V.CIUDAD = "Londres"

Q3 : RETRIEVE ( V.PROVEEDOR ) WHERE V'.V# = VP'.V#
                               AND  VP1 .P# ■ P' .P#
```

Por último separamos la consulta de dos variables que involucra a VP' y **P'**:

```
D4 : RETRIEVE INTO VP'' ( VP'.V# ) WHERE VP'.P# = P'.P#

Q4 : RETRIEVE ( V.PROVEEDOR ) WHERE V'.V# - VP''.V#
```

Por lo tanto, la consulta original QO ha sido descompuesta en tres consultas de una variable, DI, D2 y D3 (cada una es una proyección de una restricción) y dos consultas de dos variables, D4 y Q4 (cada una es una proyección de una junta). Podemos representar la situación en este punto por medio de la estructura de árbol que muestra la figura 17.3. La figura debe ser leída de la siguiente manera:

- Las consultas DI, D2 y D3 toman como entrada a las varrels P, VP y V (más precisamente, a las relaciones que son los valores actuales de las varrels P, VP y V), respectivamente, y producen como salida a P', VP' y V, respectivamente.
- La consulta D4 toma entonces como entrada a P' y VP', y produce como salida a VP''.
- Por último, la consulta Q4 toma como entrada a V y VP", y produce como salida el resul tado general requerido.

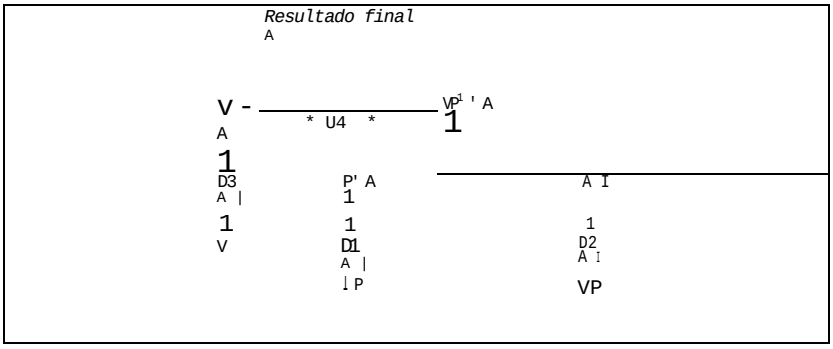


Figura 17.3 Árbol de descomposición para la consulta QO.

Observe ahora que las consultas DI, D2 y D3 son completamente independientes entre sí y pueden ser procesadas en cualquier orden (posiblemente incluso en paralelo). En forma similar, las consultas D3 y D4 pueden ser procesadas en cualquier orden, una vez que las consultas DI y D2 hayan sido procesadas. Sin embargo, las consultas D4 y Q4 ya no pueden ser descompuestas y deben ser procesadas por sustitución de tupia (lo que en realidad significa exactamente *fuerza bruta, búsqueda con índice* o *búsqueda por dispersión,* vea la sección 17.7). Por ejemplo, considere la consulta Q4. Con nuestros datos de ejemplo usuales, el conjunto de números de proveedor en el atributo VP".V# será el conjunto {VI, V2, V4}. Cada uno de estos tres valores sustituirá a VP".V#. Por lo tanto, Q4 será evaluada como si hubiera sido escrita de la siguiente manera:

```
RETRIEVE ( V. PROVEEDOR ) WHERE V'.V# = "V1 "
                          OR V .V# • "V2"
                          OR   V.V# = "V4"
```

La referencia [17.36] proporciona algoritmos para dividir la consulta original en consultas de componentes irreductibles y para seleccionar variables para la sustitución de tupias. Es en esta última selección donde reside gran parte de la optimización actual; la referencia [17.36] incluye técnicas para hacer las estimaciones de costos que manejan la selección (por lo general, pero no siempre, Ingres escoge la relación que tiene la cardinalidad más pequeña para hacer la sustitución). Los objetivos principales del proceso de optimización como un todo, son evitar tener que construir productos cartesianos y mantener al mínimo la cantidad de tupias a revisar en cada etapa.

La referencia [17.36] no explica la optimización de consultas de una variable. Sin embargo, la información relacionada con ese nivel de optimización es dada en el artículo introductorio de Ingres [7.11]. En esencia, es similar a la función equivalente en otros sistemas, que implicad uso de la información estadística mantenida en el catálogo y la selección de una ruta de acceso específica (por ejemplo, la dispersión o un índice) para revisar los datos dependiendo de cómo estén guardados.

La referencia [17.37] presenta alguna evidencia experimental (mediciones comparativas de un conjunto de consultas) que sugieren que las técnicas de optimización descritas anteriormente son básicamente firmes y bastante efectivas en la práctica. Las siguientes son algunas conclusiones específicas de ese artículo:

1. La separación es el mejor movimiento inicial.
2. Si *es necesario* realizar primero la sustitución de tupia, entonces la mejor selección de varia ble a sustituir es una variable de junta.
3. Una vez que la sustitución de tupia se ha aplicado a una variable de una consulta de dos variables, una táctica excelente es construir un índice o hacer la dispersión "sobre la mar cha" (en caso necesario) en el atributo de junta de la otra relación. De hecho, Ingres aplica frecuentemente esta táctica.

17.7 IMPLEMENT ACIÓN DE LOS OPERADORES RELACIÓNALES

Ahora presentamos una breve descripción de algunos métodos directos para implementar algunos de los operadores relaciónales, en particular el de junta. Nuestra razón principal para incluir este material es eliminar simplemente cualquier aire de misterio que haya quedado y pudiera

rodear todavía al proceso de optimización. Los métodos a explicar corresponden a lo que en la sección 17.3 hemos llamado "procedimientos de implementación de bajo nivel". *Nota:* Algunas técnicas de implementación mucho más sofisticadas están descritas en los comentarios de algunas de las referencias que aparecen al final del capítulo.

Por razones de simplicidad, damos por hecho que las tupias y las relaciones están almacenadas físicamente como tales. Los operadores que consideramos son: proyectar, juntar y resumir; tomamos a "resumir" para que incluya los dos siguientes casos.

1. Resumir sin considerar ningún atributo.
2. Resumir por al menos un atributo.

El caso 1 es directo. Involucra básicamente la revisión de la relación completa sobre la que se va a hacer el resumen; excepto que si el atributo a ser resumido (por ejemplo, promediado) está indexado, sería posible calcular el resultado directamente desde el índice sin tener que acceder a la propia relación [17.35]. Por ejemplo, la expresión

```
SUMMARIZE VP ADD AVG ( CANT ) AS PROMCANT
```

puede ser evaluada revisando el índice CANT (suponiendo que exista) sin tocar para nada los *propios* envíos. Un comentario similar se aplica si se reemplaza AVG por COUNT o SUM (para COUNT servirá cualquier índice). En lo que se refiere a MAX y MIN, el resultado puede ser encontrado con *un solo acceso* a la última entrada del índice (para MAX) o a la primera (para MIN), suponiendo nuevamente que exista un índice para el atributo relevante.

Para el resto de esta sección tomaremos "resumir" para que signifique específicamente el caso 2. Éste es un ejemplo del caso 2:

```
SUMMARIZE VP PER P { P# } ADD SUM ( CANT ) AS CANTOT
```

Desde el punto de vista del usuario, proyectar, juntar y resumir del caso 2 son por supuesto muy diferentes entre sí. Sin embargo, desde un punto de vista de implementación tienen ciertas similitudes, ya que en cada uno de los casos el sistema necesita agrupar tupias con base en valores comunes para los atributos especificados. En el caso de la proyección, dicho agrupamiento permite que el sistema elimine duplicados; en el caso de la junta, permite encontrar tupias coincidentes y en el caso del resumen, permite calcular los valores agregados individuales (es decir, por grupo). Existen varias técnicas para realizar tal agrupamiento:

1. Fuerza bruta.
2. Búsqueda con índice.
3. Búsqueda con dispersión.
4. Mezcla.
5. Dispersión.
6. Combinaciones de lo anterior.

Las figuras 17.4 a 17.8 dan procedimientos en seudocódigo para el caso específico de una junta (la proyección y el resumen se dejan como ejercicios). La notación usada en esas figuras es la siguiente. Primero, *R* y *S* son las relaciones a juntar y *C* es su atributo común (posiblemente

compuesto). Suponemos que es posible acceder a las tupias de *R* y *S,* una por una, en alguna secuencia y que denotamos esas tupias en esa secuencia como *R[\], R[2],..., R[m]* y *S[i], S[2],...,* S[n], respectivamente. Usamos la expresión *R[i] * S\j]* para denotar a la tupia juntada que se forma a partir de las tupias *R[i] y S\j].* Por último, nos referimos a *R* como la relación **externa** y a *S* como la relación **interna** (ya que controlan los ciclos externo e interno, respectivamente).

Fuerza bruta

La fuerza bruta es lo que podría decirse el caso "llano" donde se inspeccionan todas las combinaciones posibles de tupias (es decir, cada tupia de *R* es examinada en conjunción con cada tupia de *S;* como lo indica la figura 17.4). *Nota:* En ocasiones, a la fuerza bruta se le llama "ciclos anidados"; aunque este nombre causa confusión, ya que los ciclos anidados de hecho están involucrados en todos los algoritmos.

```
do  i := 1 to m                                          /* ciclo externo  */
    do j := 1 t  n ;                                     /* ciclo interno  *1
      if R[i]  .  = S[j  .C then
      añadir   l  tupia  juntada        "      al  resultado ;
    end ;
en  1
d
```

Figura 17.4 Fuerza bruta.

Examinemos los costos asociados con el enfoque de fuerza bruta. *Nota:* Aquí limitamos nuestra atención solamente al costo de E/S, aunque en la práctica también pudieran ser importantes otros costos (por ejemplo, costos de CPU).

En primer lugar, el enfoque requiere claramente un total de *m* + (m * *rí)* operaciones de lectura de tupias, pero ¿qué hay acerca de las escrituras de tupias?; es decir, ¿cuál es la cardinalidad del resultado juntado? (La cantidad de escrituras de tupias será igual a esa cardinalidad si el resultado se escribe de nuevo en el disco).

- En el importante caso especial de una junta de muchos a uno (en particular una junta de clave externa con una clave candidata coincidente), es claro que la cardinalidad del resul tado es igual a la cardinalidad —es decir, *m* o *n*— de i? o de *S,* dependiendo de cuál repre sente el lado de clave externa de la junta.
- Consideremos ahora el caso más general de una junta de muchos a muchos. Hagamos que *dCR* sea la cantidad de valores distintos del atributo de junta *C* en la relación *R* y que *dCS* esté definido en forma similar. Si suponemos una *distribución uniforme de valores* (con el fin de que cualquier valor dado de *C* en la relación *R* pueda ocurrir como cualquier otro), entonces para una tupia dada de *R* habrá *nldCS* tupias de *S* con el mismo valor para C que esa tupia; por lo tanto, la cantidad total de tupias en la junta (es decir, la cardinalidad del resultado) será (m * *n)ldCS.* O bien, si comenzamos considerando una tupia dada de *S* en vez de *R,* la cantidad total será (n * *m)ldCR.* Las dos estimaciones diferirán si *dCR + dCS* es decir, si hay algunos valores de *C* que existan en *R* pero no en *S,* o *viceversa,* en cuyo caso el estimado menor es el que hay que usar.

Por supuesto (como establecimos en la sección 17.2), en la práctica lo que importa es la E/S de *páginas* y no de tupias. Por lo tanto, supongamos que las tupias de *R* y *S* están guardadas *pR*

en una página y *pS* en otra, respectivamente (y de esa forma las dos relaciones ocupan *mlpR* y *nlpS* páginas, respectivamente). Es fácil ver que el procedimiento de la figura 17.4 involucrará *(m/pR) + (m * ri)lpS* lecturas de páginas. En forma alterna, si intercambiamos los papeles de *R* y *S* (haciendo que *S* sea la relación externa y *R* la interna) la cantidad de lecturas de páginas será *(n/pS) + (n* m)lpR.*

Como ejemplo, supongamos que *m* - 100, *n* = 10,000, *pR* = 1 y *pS* = 10. Entonces las dos fórmulas dan como resultado 100,100 y 1,001,000 lecturas de páginas, respectivamente. *Conclusión:* en el enfoque de fuerza bruta es necesario que se escoja la relación más pequeña de las dos como relación externa (donde *más pequeña* significa "menor cantidad de páginas").

Concluimos esta breve explicación de la técnica de fuerza bruta con la observación de que ésta debe ser considerada como el procedimiento del peor caso, ya que asume que la relación *S* no está indexada ni dispersada sobre el atributo de junta C. Los experimentos de Bitton *et al* [17.7] indican que si esa suposición es de hecho válida, las cosas mejorarán, por lo general, con la construcción de un índice o una dispersión dinámicamente, continuando con una junta con búsqueda con índice o con dispersión (vea las dos siguientes subsecciones). La referencia [17.37] apoya esta idea, como mencioné al final de la sección anterior.

Búsqueda con índice

Ahora consideramos el caso en el que existe un índice *X* sobre el atributo *S.C* de la relación interna (consulte la figura 17.5). La ventaja de esta técnica sobre la de fuerza bruta es que para una tupia dada de la relación externa *R,* podemos ir "directamente" a las tupias coincidentes de la relación interna *S,* Entonces, la cantidad total de lecturas de tupias de las relaciones *R y S* es simplemente la cardinalidad del resultado juntado; tomando el peor de los casos en el que cada lectura de tupia de *S* es de hecho una lectura de página separada, entonces la cantidad total de lecturas de páginas para *R* y *S* es *(m/pR) + (mn/dCS).*

```
/*  supong i un índice X sobre S.C */
    í
do    i   = 1 to m ;                                  /*  ciclo externo   */
      /* sean k entradas de índice X[1], .  . ., XIk] con                  */
      /* el valor del atributo indexado =  fl[i] .C                        */
      do j := 1 to k ;                                 1*  ciclo interno   */
         1* sea la tupia de S indexada por  X[j] igual a Slj] *
          añadir la tupia juntada fl[i] »   S[j] al resultado ;
      end;
en  !
d
```

Figura **17.5** Búsqueda con índice.

Sin embargo, si resulta que la relación *S* está guardada en secuencia por los valores del atributo de junta C, la cifra de lecturas de páginas se reduce a *(tnlpR) + (mnldCS)lpS.* Si tomamos los mismos valores de ejemplo que en el caso anterior (m = 100, *n*= 10,000,*pR* = *l,pS*= 10) y suponemos que *dCS* = 100, las dos fórmulas dan como resultado 10,100 y 1,100, respectivamente. La diferencia entre estas dos cifras resalta claramente la importancia de conservar las relaciones guardadas en una secuencia física "buena" [17.9].

Sin embargo, debemos por supuesto incluir la sobrecarga del acceso al propio índice *X.* La suposición del peor caso es que cada tupia de *R* requiere una búsqueda de índice "nueva" para

encontrar las tupias coincidentes de *S*, lo que implica leer una página de cada nivel del índice. Para un índice de x niveles, esto añadirá unas lecturas de páginas *mx* adicionales a la lectura general de páginas. En la práctica, *x* será típicamente 3 o menos (además, es muy probable que el nivel superior del índice resida en un buffer de memoria principal a lo largo del proceso, lo que por lo tanto reduce aún más la cifra de lecturas de páginas).

Búsqueda con dispersión

La búsqueda con dispersión es similar a la búsqueda con índice, con excepción de que la "ruta de acceso rápida" a la relación interna *S* sobre el atributo de junta 5. C, se hace con dispersión en lugar de con un índice (consulte la figura 17.6). Las derivaciones de las estimaciones de costos para este caso se dejan como ejercicio.

```
/* suponga una tabla   dispersión H  sobre S.  */
   de                                        C
do i := 1 to m ;                              /  ciclo  externo */
   k := hash (R[i].C)   j
   /* sean h tupias     S[1], . . .,  S[h)   guardadas  sn H[k]  */
   do j := 1 to h ;                           /  ciclo  interno */
     if 81}].C = fl[i]  .C then
     añadir la tupia    juntada R[i]   '      al  resultado ;
   end ;
en
d
```

Figura 17.6 Búsqueda con dispersión.

Mezcla

La técnica de mezcla supone que las dos relaciones *R* y *S* están físicamente guardadas en secuencia por los valores del atributo de junta *C*. Si este es el caso, las dos relaciones pueden ser revisadas en secuencia física, las dos revisiones pueden estar sincronizadas y la junta completa puede ser realizada en una sola pasada sobre los datos (al menos esto es cierto si la junta es de uno a muchos, ya que podría no ser tan cierto para el caso de muchos a muchos). Dicha técnica es incuestionablemente óptima, ya que cada página es accedida una sola vez (consulte la figura 17.7). En otras palabras, la cantidad de páginas leídas es sólo *(m/pR) + (n/pS)*. Se desprende que:

- El agrupamiento físico de los datos relacionados lógicamente es uno de los factores de de sempeño más crítico de todos; es decir, es muy necesario que los datos estén agrupados en esa forma para que concuerden con las juntas que son más importantes para la empresa [17.9].
- En ausencia de tal agrupamiento, a menudo es buena idea ordenar una o ambas relaciones en tiempo de ejecución y luego hacer una junta de mezcla (por supuesto, el efecto de tal ordenamiento es precisamente producir dinámicamente el agrupamiento deseado). A esta técnica se le llama, con toda razón, **ordenamiento/mezcla** [17.10].

Vea la referencia [17.35] para una mejor explicación.

```
/  suponga que R y S están ordenadas   3or el atributo C ;        */
/  el siguiente código supone que  la    junta de muchos a muchos ;*/
/  el caso más simple de muchos a  un  se deja como ejercicio     */
r  :■ 1 ;
s  :■ 1 ;
d  while r < m and s < n ;                          /* ciclo externo */
   v := R[r].C ;
   do j := s by 1 while S[j).C <   v ;
   end ;
   s := j ; do ; := s by 1 while   v ;        / * ciclo interno principal */
       do i := r by 1 while fl[i].  C = v ;
         añadir la tupia juntada   mi  • S[j\ al resultado ;
       end ;
   end ;
   s := j ;
   do i := r by 1 while R[i].C =
   end ;
   r := i ;
end ;
```

Figura 17.7 Mezcla (caso de muchos a muchos).

Dispersión

Al igual que la técnica de mezcla que acabo de explicar, la técnica de dispersión requiere de una sola pasada sobre cada una de las dos relaciones (consulte la figura 17.8). La primera pasada construye una tabla de dispersión para la relación *S* sobre los valores del atributo de junta *S.C;* las entradas de esa tabla contienen el valor del atributo de junta —probablemente también otros valores de atributo— y un apuntador hacia la tupia correspondiente en el disco. La segunda pasada revisa la relación *R* y aplica la misma función de dispersión al atributo de junta *R.C.* Cuando una tupia *R* colisiona en la tabla de dispersión con una o más tupias de *S,* el algoritmo verifica si los valores de *R.C y S.C* son iguales, y de ser así, genera la tupia juntada adecuada. La gran ventaja de esta técnica sobre la de mezcla es que las relaciones *R y S* no necesitan estar guardadas en ningún orden específico y no es necesario ningún ordenamiento.

```
/*  construye la tabla de dispersión   sobre S  C *J
            H                                .
    k :•  1 to n ; {S[j].C) ; a la      tabla d  dispersión ;
    añad  hash entrada H[k] de la              e
do  í
      sin
  .
    ahor  hace  búsqueda con dispersión  sobre R */
    j
```

Figura 17.8 Dispersión.

Así como hicimos con la técnica de búsqueda con dispersión, dejamos como ejercicio derivar las estimaciones del costo para este enfoque.

17.8 RESUMEN

Comenzamos diciendo que la optimización representa un *reto* y una *oportunidad* para los sistemas relaciónales. De hecho, la posibilidad de optimización es una **ventaja** de esos sistemas por varias razones, ya que un sistema relacional con un buen optimizador puede superar muy bien a un sistema no relacional. Nuestro ejemplo introductorio dio algunas ideas del tipo de mejoras que pueden ser logradas (un factor de 10,000 a 1 en ese caso particular). Las cuatro grandes etapas de la optimización son:

- Convertir la consulta en alguna **forma interna** (por lo general un **árbol de consulta** o **árbol de sintaxis abstracto,** aunque esa representación puede ser considerada sólo como una forma interna del álgebra relacional o del cálculo relacional);
- Convertir a una **forma canónica** usando varias **leyes de transformación;**
- Seleccionar los **procedimientos de bajo nivel** candidatos para la implementación de las di versas operaciones de la representación canónica de la consulta;
- Generar **planes de consulta** y elegir el más barato, usando **fórmulas de costo** y conoci miento a partir de **estadísticas de la base de datos.**

Luego tratamos las leyes generales **distributiva, conmutativa** y **asociativa** y su aplicabilidad a los operadores relaciónales —tales como la junta (y también su aplicabilidad a los operadores **aritméticos, lógicos** y de **comparación)**— y mencionamos otra ley general llamada **idempotencia.** También tratamos algunas transformaciones específicas para los operadores **restricción** y **proyección.** Luego presentamos la idea importante de las transformaciones **semánticas;** es decir, las transformaciones basadas en el conocimiento del sistema de las **restricciones de integridad.**

A manera de ilustración, mencionamos algunas de las estadísticas mantenidas por los productos **DB2** e **Ingres.** Luego describimos una estrategia de "divide y vencerás" llamada **descomposición de la consulta** (la cual fue presentada con el prototipo de Ingres) y mencionamos que tal estrategia puede ser muy atractiva en un ambiente de procesamiento en paralelo o distribuido.

Por último, examinamos determinadas **técnicas de implementación** para algunos de los operadores relaciónales (en especial el de **junta).** Presentamos algoritmos en seudocódigo para cinco técnicas de junta **(fuerza bruta, búsqueda con índice, búsqueda con dispersión, mezcla** (incluyendo **ordenamiento/mezcla)** y **dispersión)** y consideramos brevemente los costos asociados con esas técnicas.

En conclusión, debemos mencionar que muchos de los productos actuales incluyen —desgraciadamente— determinados **inhibidores de optimización,** de lo cual deberán estar conscientes los usuarios (aunque en la mayoría de los casos, poco es lo que pueden hacer). Un inhibidor de optimización es una característica del sistema en cuestión que impide que el optimizador haga el buen trabajo que podría hacer (es decir, si esa característica no estuviera). Los inhibidores en cuestión incluyen a las *filas duplicadas* (vea la referencia [5.6]), la *lógica de tres valores* (vea el capítulo 18) y la *implementación de la lógica de tres valores en el SQL* (vea las referencias [18.6] y [18.10]).

EJERCICIOS

17.1 Algunos de los siguientes pares de expresiones de la base de datos de proveedores, partes y proyectos son equivalentes y otros no. ¿Cuáles son los pares equivalentes?

```
al.  V JOIN (  ( P JOIN Y ) WHERE CIUDAD =  'Londres'  )

a2.   ( P WHERE CIUDAD =  'Londres'  ) JOIN  (  Y JOIN V  )

bl.  (  V MINUS  (  (  V JOIN VPY  )  WHERE  P# = P#  (  'P2'  )  )
                 {  V#,  PROVEEDOR,  STATUS,  CIUDAD  }  )  {  V#,  CIUDAD  }

b2.  V { V#,  CIUDAD } MINUS
       (  V { V#,  CIUDAD } JOIN
         ( VPY WHERE P# = P# ( 'P2' ) ) ) { V#, CIUDAD }

Cl.  ( V { CIUDAD } MINUS P { CIUDAD } ) MINUS Y { CIUDAD }

c2.  ( V { CIUDAD } MINUS Y { CIUDAD } )

        MINUS ( P { CIUDAD } MINUS Y { CIUDAD } )

di. ( Y { CIUDAD } INTERSECT ? { CIUDAD } ) UNION ( V { CIUDAD } ) d2.

Y { CIUDAD } INTERSECT ( V { CIUDAD } UNION P { CIUDAD } )

el. ( ( VPY WHERE V# = V# ( 'V1' ) )
                      UNION ( VPY WHERE P# = P# ( 'P1' ) ) )
     INTERSECT ( ( VPY WHERE Y# = Y# ( 'Y1 ' ) )
                      UNION ( VPY WHERE V# = V# ( 'V1' ) ) )

e2. ( VPY WHERE V# = V# ( 'V1' ) ) UNION
     ( ( VPY WHERE P# = P# ( 'P1' ) ) INTERSECT (
       VPY WHERE Y# = Y# ( ' Y1 ' ) ) )

fl.  (V WHERE CIUDAD = 'Londres' ) UNION ( V WHERE STATUS > 10 )

f2. V WHERE CIUDAD ■ 'Londres' AND STATUS > 10

gl. ( V { V# } INTERSECT ( VPY WHERE Y# = Y# ( 'Y1' ) ) { V# } )
      UNION ( V WHERE CIUDAD = 'Londres' ) { V# }

g2. V { V# } INTERSECT ( ( VPY WHERE Y# = Y# ( ' Y1 ' ) ) { Vtt }
                  UNION ( V WHERE CIUDAD ■ 'Londres' ) { V# } )

hl. ( VPY WHERE Y# = Y# ( 'Y1' ) ) { V# }
      MINUS ( VPY WHERE P# = P# ( ' P1 ' ) ) { V# }

h2. ( ( VPY WHERE Y# = Y# ( 'Y1' ) )

      MINUS ( VPY WHERE P# = P# ( 'P1' ) ) ) { V# }

il.  V JOIN ( P { CIUDAD } MINUS Y { CIUDAD } )

¡2.  ( V JOIN P { CIUDAD } ) MINUS ( V JOIN Y { CIUDAD } )
```

17.2 Muestre que la junta, la unión y la intersección son conmutativas y que la diferencia no lo es.

17.3 Muestre que la junta, la unión y la intersección son asociativas y la diferencia no lo es.

17.4 Muestre que:

a. La unión se distribuye sobre la intersección;

b. La intersección se distribuye sobre la unión.

17.5 Muestre que para toda *A* y *B:*

a. `A UNION ( A INTERSECT B ) = A ;`

b. `A INTERSECT ( A UNION B ) = A ;`

Nota: Estas dos leyes son llamadas de **absorción.** Al igual que las leyes de idempotencia y conmutativa (entre otras), también pueden resultar útiles para propósitos de optimización.

17.6 Muestre que:

a. La restricción es distributiva incondicionalmente sobre la unión, la intersección y la diferencia, y condicionalmente sobre la junta;

b. La proyección es distributiva incondicionalmente sobre la unión y la intersección, lo es condi cionalmente sobre la junta, y no es distributiva sobre la diferencia.

Indique las condiciones relevantes en los casos condicionales.

17.7 Amplíe las reglas de transformación de la sección 17.4 para que tomen en cuenta a EXTEND y SUMMARIZE.

17.8 ¿Puede encontrar alguna regla de transformación útil para la operación de la división relacional?

17.9 Dé un conjunto de reglas de transformación adecuadas para las expresiones condicionales que involucran a AND, OR y NOT. Un ejemplo de estas reglas podría ser la "conmutatividad de AND"; es decir, *A* AND *B* es lo mismo que *B* AND *A.*

17.10 Amplíe las respuestas al ejercicio anterior para que incluyan expresiones lógicas que in volucren a los cuantificadores EXISTS y FORALL. Un ejemplo de esta regla podría ser la regla dada en el capítulo 7 (sección 7.2) que permite que una expresión que involucra a FORALL se con vierta en otra que involucre la negación de EXISTS.

17.11 Ésta es una lista de restricciones de integridad para la base de datos de proveedores, partes y proyectos (extraída de los ejercicios del capítulo 8):

- Las únicas ciudades válidas son Londres, París, Roma, Atenas, Oslo, Estocolmo, Madrid y Amsterdam.
- Dos proyectos no pueden estar ubicados en la misma ciudad.
- Como máximo, un proveedor puede estar ubicado en Atenas en cualquier momento.
- Ningún envío puede tener una cantidad mayor al doble del promedio de todas esas cantidades.
- El proveedor con el status más alto no debe estar en la misma ciudad que el proveedor con el status más bajo.
- Todo proyecto debe encontrarse en una ciudad en la cual exista al menos un proveedor para ese proyecto.
- Como mínimo, debe existir una parte roja.
- El status promedio de los proveedores debe ser mayor que 18.
- Todo proveedor de Londres debe proporcionar la parte P2.
- Al menos una parte roja debe pesar menos de 50 libras.
- Los proveedores de Londres deben proporcionar más tipos de partes que los de Pans.
- Los proveedores de Londres deben proporcionar más partes en total que los de París.

Y estas son algunas consultas de ejemplo para la base de datos:

a. Obtener los proveedores que no proporcionan la parte P2.
b. Obtener los proveedores que no abastecen un proyecto en la misma ciudad que el proveedor.
c. Obtener los proveedores que no proporcionen la menor cantidad de tipos de partes.
d. Obtener los proveedores de Oslo que proporcionan al menos dos partes distintas de París a al menos dos proyectos distintos de Estocolmo.
e. Obtener pares de proveedores coubicados que proporcionan pares de partes coubicadas.
f. Obtener pares de proveedores coubicados que abastecen pares de proyectos coubicados.
g. Obtener partes proporcionadas al menos para un proyecto sólo por proveedores que no estén en la misma ciudad que el proyecto.
h. Obtener proveedores que no proporcionen más tipos de partes.

Use las restricciones de integridad para transformar estas consultas en formas más sencillas (todavía en lenguaje natural, ya que aún no le pedimos que realice este ejercicio *formalmente).*

17.12 Investigue cualquier DBMS que tenga disponible. ¿Ese sistema realiza alguna transformación de expresiones? (no todos lo hacen). De ser así, ¿qué transformaciones realiza?, ¿realiza alguna trans formación *semántica'}*

17.13 Intente el siguiente experimento: tome una consulta sencilla —digamos "obtener los nombres de los proveedores que proporcionan la parte P2"— y presente esa consulta en tantas formas diferen tes como se le ocurra en cualquier lenguaje de consulta que tenga disponible (probablemente SQL). Cree y pueble una base de datos de prueba adecuada que ejecute las diferentes versiones de la consul ta y mida los tiempos de ejecución. Si esos tiempos varían significativamente, ya tiene una evidencia empírica de que el optimizador no está haciendo un buen trabajo de transformación de expresiones. Repita el experimento con varias consultas diferentes. De ser posible repítalo también con varios DBMS diferentes.

Nota: Por supuesto, todas las versiones diferentes de la consulta deberán dar el mismo resultado. De no ser así es probable que haya cometido un error o quizá se deba a un error del optimizador; de ser así, ¡repórtelo al fabricante!

17.14 Investigue cualquier DBMS que tenga disponible. ¿Mantiene el sistema alguna estadística de la base de datos? (No todos los hacen). De ser así, ¿cuáles son?, ¿cómo se actualizan?, ¿en forma dinámica o por medio de alguna utilería? De ser así, ¿qué utilería se usa?, ¿qué tan frecuentemente se ejecuta?, ¿qué tan selectiva es en términos de las estadísticas específicas que puede actualizar en cual quier ejecución específica?

17.15 Vimos en la sección 17.5 que entre las estadísticas de la base de datos mantenidas por DB2 están los valores segundo más alto y segundo más bajo de cada columna de cada tabla base. ¿Por qué cree usted que sean los *segundos* más alto y más bajo?

17.16 Varios productos comerciales permiten que el usuario proporcione pistas al optimizador. Por ejemplo, en DB2 la especificación OPTIMIZE FOR *n* ROWS en una declaración de cursor SQL, sig nifica que el usuario espera recuperar no más de *n* filas por medio del cursor en cuestión (es decir, ejecutar FETCH sobre el cursor no más de *n* veces). En ocasiones, dicha especificación puede hacer que el optimizador elija una ruta de acceso que es más eficiente al menos para el caso donde el usuario ejecuta a FETCH no más de *n* veces. ¿Cree usted que tal pista sea buena idea? Justifique su respuesta.

17.17 Imagine un conjunto de procedimientos de implementación para las operaciones de restricción y proyección (similares a los procedimientos que esbozamos para la junta de la sección 17.7). Derive un conjunto adecuado de fórmulas de costo para esos procedimientos. Suponga que la E/S de pági nas es la única cantidad que interesa; es decir, no intente incluir el costo de la CPU u otros costos en las fórmulas. Establezca y justifique cualquier otra suposición que haga.

REFERENCIAS Y BIBLIOGRAFÍA

El campo de la optimización es inmenso y se está desarrollando todo el tiempo (de hecho, está llegando a ser más importante que nunca, gracias al creciente interés en los sistemas de apoyo a la toma de decisiones; vea el capítulo 21). Para dar un ejemplo, más del 50 por ciento de los artículos presentados en cada una de las conferencias anuales ACM SIGMOD durante los últimos años, han estado relacionados con algún tipo de optimización. La siguiente lista representa una selección relativamente pequeña entre la amplia literatura de este tema. Está dividida burdamente en grupos, de la siguiente manera:

- Las referencias [17.1] a [17.7] proporcionan introducciones o panoramas generales del pro blema de optimización en general.
- Las referencias [17.8]a[17.17]se concentran en la implementación eficiente de alguna ope ración relacional específica, tal como la junta o el resumen.
- Las referencias [17.18] a [17.33] describen una variedad de técnicas basadas en la trans formación de expresiones, como se trata en la sección 17.4; en particular, las referencias [17.27] a [17.30] consideran las transformaciones *semánticas.*
- Las referencias [17.34] a [17.45] explican las técnicas usadas en System R, DB2 e Ingres, así como el problema general de la optimización de consultas que involucran subconsultas anidadas al estilo SQL.
- Las referencias [17.46] a [17.61] tratan un conjunto diverso de técnicas, trucos e ideas para investigación futura. En particular, las referencias [17.58] a [17.61] consideran el impacto del procesamiento paralelo en el problema de la optimización.

Nota: Se excluyen deliberadamente las publicaciones sobre optimización en sistemas distribuidos. Vea el capítulo 20.

17.1 Won Kim, David S. Reiner y Don S. Batory (eds.): *Query Processing in Database Systems.* New York, N.Y.: Springer Verlag (1985).

Este libro es una antología de artículos sobre el tema general de procesamiento de consultas (no sólo sobre la optimización). Consiste en un artículo introductorio de Jarke, Koch y Schmidt (similar pero no idéntico a la referencia [17.3]), seguido por un grupo de artículos que explican el procesamiento de consultas en una diversidad de contextos: bases de datos distribuidas, sistemas heterogéneos, actualización de vistas (la referencia [9.11] es el único artículo en esta sección), aplicaciones no tradicionales (por ejemplo, CAD/CAM), optimización de varias instrucciones (vea la referencia [17.49]), máquinas de bases de datos y el diseño de bases de datos físicas.

17.2 IEEE: *Database Engineering 5,* No. 3: Special Issue on Query Optimization (septiembre, 1982).

Contiene 13 artículos coitos (de ambientes académicos y comerciales) sobre diversos aspectos de la optimización de consultas.

17.3 Matthias Jarke y Jürgen Koch: "Query Optimization in Database Systems", *ACM Comp. Surv. 16,* No. 2 (junio, 1984).

Es un tutorial excelente. El artículo proporciona un marco general de trabajo para la evaluación de consultas; muy similar al de la sección 17.3 de este capítulo, pero basado en el cálculo relacional en lugar del álgebra. Luego trata una gran cantidad de técnicas de optimización dentro de ese marco de trabajo: transformaciones sintácticas y semánticas, implementación de operaciones de bajo nivel, así como algoritmos para generar planes de consulta y para seleccionarlos. Proporciona un amplio conjunto de reglas de transformación sintáctica para las expresiones de

cálculo. También incluye una amplia bibliografía (sin comentarios); sin embargo, debe tener presente que la cantidad de artículos sobre el tema publicados desde 1984 es probablemente de un orden de magnitud mayor que la cantidad anterior a esa fecha (vea la referencia [17.4]).

El artículo también explica brevemente algunos otros temas relacionados: la optimización de los lenguajes de consulta de alto nivel (es decir, lenguajes que son más poderosos que el álgebra o el cálculo), la optimización en un ambiente de bases de datos distribuidas y el papel de las máquinas de bases de datos con respecto a la optimización.

17.4 Götz Graefe: "Query Evaluation Techniques for Large Databases", *ACM Comp. Surv. 25,* No. 2 (junio, 1993).

Otro tutorial excelente (más reciente) con una amplia bibliografía. Citando el resumen: "este estudio proporciona una base para el diseño e implementación de propiedades para la ejecución de consultas... describe un amplio arreglo de técnicas de evaluación de consultas prácticas... incluyendo la ejecución iterativa de planes complejos de evaluación de una consulta, la dualidad de los algoritmos de concordancia de conjuntos basados en el ordenamiento y la dispersión, los tipos de ejecución de consultas en paralelo y su implementación, así como operadores especiales para los dominios de aplicaciones de bases de datos emergentes". Recomendable.

17.5 Frank P. Palermo: "A Data Base Search Problem", en Julius T. Tou (ed.), *Information Systems: COINS IV.* New York, N.Y.: Plenum Press (1974).

Es uno de los primeros artículos sobre optimización (de hecho, un clásico). Parte de una expresión cualquiera del cálculo relacional, el artículo usa primero el algoritmo de reducción de Codd para reducir esa expresión a una expresión algebraica equivalente (vea el capítulo 7) y luego presenta varias mejoras a ese algoritmo, entre ellas las siguientes:

- Ninguna tupia es recuperada más de una vez.
- Tan pronto como se recupera la tupia, los valores innecesarios se descartan de la tupia (donde los "valores innecesarios" son los de los atributos no referidos en la consulta o los usados únicamente para propósitos de restricción). Este proceso es equivalente a la proyección de la relación sobre los atributos "necesarios" y por lo tanto, no sólo reduce el espacio requerido para cada tupia sino que también reduce la cantidad de tupias que necesitan ser conservadas (en general).
- El método usado para construir la relación resultante está basado en un principio de mínimo crecimiento, por lo que el resultado tiende a crecer lentamente. Esta técnica tiene el efecto de reducir la cantidad de comparaciones involucradas y la cantidad de almacenamiento inter medio requerido.
- Se emplea una técnica eficiente en la construcción de juntas, que involucra (a) la factorización dinámica de valores usados en los términos de la junta (tales como V.V# = VP. V#) en *semijuntas,* que en efecto son un tipo de índice secundario construido dinámicamente (las semijuntas de Palermo no son lo mismo que las semijuntas del capítulo 6) y (b) el uso de una representa ción interna de cada junta —llamada una *junta indirecta*— la cual utiliza el ID de tupia in terno para identificar las tupias que participan en la junta. Estas técnicas están diseñadas para reducir la cantidad de revisiones necesarias para la construcción de la junta, lo que asegura (para cada término de la junta) que las tupias involucradas estén ordenadas en forma lógica sobre los valores de los atributos de la junta. También permite la determinación dinámica de una "mejor" secuencia en la cual es posible acceder a las relaciones requeridas.

17.6 Jim Gray (ed.): *The Benchmark Handbook for Database and Transaction Processing Systems* (2a. edición). San Francisco, Calif.: Morgan Kaufmann (1993).

El TPC (Consejo de Procesamiento de Transacciones) es un órgano independiente que ha producido varias pruebas estándares de la industria a través de los años y este libro incluye información

detallada sobre esas pruebas en particular (y también sobre otras). *TPC-A* es una prueba que pretende medir el desempeño OLTP (procesamiento de transacciones en línea). *TPC-B* es una versión de TPC-A que mide el desempeño del DBMS y el sistema operativo subyacente, ignorando consideraciones que tienen que ver con la comunicación del usuario y cosas parecidas. *TPC-C* está modelada bajo un sistema de pedidos/captura de datos (y de hecho, sobrepasa en gran forma a TPC-A). *TPC-D* mide el desempeño de soporte a la toma de decisiones; involucra un conjunto de 17 consultas SQL bastante complejas (y por lo tanto, es el único de los cuatro que en realidad se enfoca en la calidad del optimizador como tal).

Nota: Los fabricantes compiten continuamente entre sí sobre su desempeño en las pruebas TPC. Sin embargo, es conveniente decir que hay que tener precaución en la interpretación de lo que dicen al respecto en los anuncios, ya que loS vendedores pueden y emplean todo tipo de trucos y técnicas para elevar sus cifras TPC al máximo posible. *Tenga cuidado al comprar.*

17.7 Dina Bitton, David J. DeWitt y Carolyn Turbyfill: "Benchmarking Database Systems: A Sys tematic Approach", Proc. 9th Int. Conf. on Very Large Data Bases, Florence, Italy (octubre-noviembre, 1983).

Es el primer artículo que describe lo que ahora se le llama comúnmente "la prueba Wisconsin" (debido a que fue desarrollada por los autores del artículo en la Universidad de Wisconsin). La prueba define un conjunto de relaciones con valores de atributos especificados precisamente y luego mide el desempeño de determinadas operaciones algebraicas especificadas precisamente sobre esas relaciones (por ejemplo, diversas proyecciones que involucran diferentes grados de duplicación en los atributos sobre los cuales se toman las proyecciones). Por lo tanto, representa una prueba sistemática de la efectividad del optimizador sobre esas operaciones fundamentales. Vea también la referencia [17.6].

17.8 S. Bing Yao: "Optimization of Query Evaluation Algorithms", ACM *TODS 4,* No. 2 (junio, 1979).

Desarrolla un modelo general de procesamiento de consultas que incluye muchos algoritmos familiares como casos especiales. El modelo incluye el siguiente conjunto de operaciones de bajo nivel, junto con un conjunto asociado de fórmulas de costo:

- Indexado de restricciones
- Intersección
- Rastreo secuencial
- Filtro de restricciones
- Ordenamiento
- Proyección
- Indexado de junta
- Acceso a registro
- Rastreo de enlaces
- Filtro de junta
- Concatenación

Un algoritmo de procesamiento de consultas dado —expresado en términos de estas operaciones de bajo nivel— puede ser evaluado de acuerdo con las fórmulas de costo. El artículo identifica diversas clases de algoritmos de procesamiento de consultas y asigna una fórmula de costos para cada clase. Entonces, el problema de optimización de consultas se convierte en el problema de resolver un conjunto simple de ecuaciones para encontrar un costo mínimo y luego seleccionar la clase de algoritmo que corresponde a ese costo mínimo.

17.9 M. W. Blasgen y K. P. Eswaran: "Storage and Access in Relational Databases", *IBM Sys. J. 16,* No. 4 (1977).

Compara varias técnicas para el manejo de consultas que involucran operaciones de restricción, proyección y junta con base en su costo de E/S de disco. Las técnicas en cuestión son básicamente las implementadas en el System R [17.34].

17.10 T. H. Merrett: "Why Sort/Merge Gives the Best Implementation of the Natural Join", *ACM SIGMOD Record 13,* No. 2 (enero, 1983).

Presenta un conjunto de argumentos intuitivos para apoyar el enunciado planteado en el título. El argumento es esencialmente que:

a. La propia operación de junta será más eficiente si las dos relaciones están ordenadas sobre los valores del atributo de junta (ya que en ese caso —como vimos en la sección 17.7— la mezcla es la técnica obvia y cada página de datos será recuperada una sola vez, lo cual es claramente óptimo).
b. El costo de ordenar las relaciones en esa secuencia deseada, en una máquina lo suficiente mente grande, será probablemente menor al costo de cualquier esquema que tome en cuenta el hecho de que no están ordenadas.

Sin embargo, el autor admite que podría haber algunas excepciones a esta posición contenciosa. Por ejemplo, una de las relaciones podría ser lo suficientemente pequeña —podría por ejemplo ser el resultado de una operación de restricción anterior— que dirigiera el acceso a la otra relación por medio de un índice o una dispersión podría ser más eficiente que el ordenamiento de esa relación. Las referencias [17.11] a [17.13] dan ejemplos adicionales de casos en donde el ordenamiento/mezcla no es probablemente la mejor técnica en la práctica.

17.11 Giovanni Maria Sacco: "Fragmentation: A Technique for Efficient Query Processing", *ACM TODS 11,* No. 2 (junio, 1986).

Presenta un método de "divide y vencerás" para realizar juntas dividiendo recursivamente las relaciones a ser juntadas en restricciones disjuntas ("fragmentos") y realizando una serie de rastreos secuenciales en esos subconjuntos. A diferencia del ordenamiento/mezcla, la técnica no requiere que las relaciones estén ordenadas primero. El artículo muestra que la técnica de fragmentación siempre se desempeña mejor que el ordenamiento/mezcla en el caso en el que éste requiera que ambas relaciones sean ordenadas primero y (por lo general) se desempeña mejor en el caso en que el ordenamiento/mezcla requiera que sólo una relación —la más grande— sea ordenada primero. El autor dice que la técnica también puede ser aplicada a otras operaciones, tales como la intersección y la diferencia.

17.12 Leonard D. Shapiro: "Join Processing in Database Systems with Large Main Memories", *ACM TODS 11,* No. 3 (septiembre, 1986).

Presenta tres algoritmos de junta con dispersión, uno de los cuales es "especialmente eficiente cuando la memoria principal disponible es una fracción significativa del tamaño de una de las relaciones a ser juntadas". El algoritmo trabaja dividiendo las relaciones en particiones disjuntas (es decir, restricciones) que pueden ser procesadas en la memoria principal. El autor afirma que los métodos de dispersión están destinados a convertirse en la técnica a escoger, tomando en cuenta la tasa a la que están disminuyendo los costos de la memoria principal.

17.13 M. Negri y G. Pelagatti: "Distributive Join: A New Algorithm for Joining Relations", *ACM TODS 16,* No. 4 (diciembre, 1991).

Otro método de junta de "divide y vencerás". "[El método] está basado en la idea de que... no es necesario ordenar completamente ambas relaciones... Es suficiente con ordenar una completamente y la otra sólo parcialmente, lo que evita, por lo tanto, parte del esfuerzo de ordenamiento." El ordenamiento parcial divide la relación afectada en una secuencia de particiones no ordenadas *PI, P2,..., Pn* (algo parecido al método de Sacco [17.11], con excepción de que Sacco usa dispersión en vez de ordenamiento) con la propiedad de que MAX(P¿) < MIN(P(i+l)) para todas las *i* (1,2,..., *n~\).* El artículo dice que este método se desempeña mejor que el de ordenamiento/mezcla.

17.14 Gótz Graefe y Richard L. Cole: "Fast Algorithms for Universal Quantification in Large Data bases", *ACM TODS 20,* No. 2 (junio, 1995).

El cuantificador universal (FORALL) no es soportado directamente en SQL y por lo tanto, tampoco por ningún DBMS comercial; pero aun así es extremadamente importante en la formulación de una amplia clase de consultas. Este artículo describe y compara "tres algoritmos conocidos y uno propuesto recientemente para la división relacional [la cual representa] al operador del álgebra que comprende la cuantificación universal", y muestra también que el nuevo algoritmo se ejecuta "con la misma rapidez con que las (semi)juntas con dispersión evalúan a la cuantificación existencial sobre las mismas relaciones" (cambiando un poco las palabras). Los autores concluyen, entre otras cosas, que FORALL deberá ser soportado directamente en el lenguaje del usuario, debido a que la mayoría de los optimizadores "no reconocen las formulaciones algo indirectas que están disponibles en SQL".

17.15 Dina Bitton y David J. DeWitt: "Duplicate Record Elimination in Large Data Files", *ACM TODS 8*, No. 2 (junio, 1983).

La técnica tradicional para la eliminación de duplicados es simplemente ordenar los registros y luego hacer una revisión secuencial. Este artículo propone un enfoque alterno que tiene características de desempeño significativamente mejores cuando el archivo es grande.

17.16 David Simmen, Eugene Shekita y Timothy Malkemus: "Fundamental Techniques for Order Optimization", Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (junio, 1996).

Presenta técnicas para optimizar o evitar ordenamientos. Las técnicas, apoyadas parcialmente en el trabajo de Darwen [10.6], han sido implementadas en DB2.

17.17 Gurmeet Singh Manku, Sridhar Rajagopalan y Bruce G. Lindsay: "Approximate Medians and Other Quantiles in One Pass and with Limited Memory", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

17.18 James Miles Smith y Philip Yen-Tang Chang: "Optimizing the Performance of a Relational Algebra Database Interface", *CACM 18*, No. 10 (octubre, 1975).

Describe los algoritmos usados en la "interfaz de consulta inteligente para un álgebra relacional" (SQUIRAL). Las técnicas usadas incluyen las siguientes:

- Transformar la expresión algebraica original en una secuencia de operaciones equivalente, pero más eficientes, de acuerdo con los lincamientos que se trataron en la sección 17.4.
- Asignar operaciones distintas en la expresión transformada para distintos procesos y explotar la concurrencia y la canalización entre ellos.
- Coordinar los ordenamientos de las relaciones temporales que se pasan entre esos procesos.
- Explotar los índices e intentar localizar referencias de páginas.

Este artículo fue uno de los primeros en tratar la transformación de expresiones.

17.19 P. A. V. Hall: "Optimisation of a Single Relational Expression in a Relational Data Base System", *IBM J. R&D 20*, No. 3 (mayo, 1976).

Este artículo describe algunas de las técnicas de optimización usadas en el sistema PRTV [6.9]. El PRTV, al igual que SQUIRAL [17.18], comienza transformando la expresión algebraica dada en una forma más eficiente antes de evaluarla (este artículo también fue uno de los primeros en tratar la transformación de expresiones). Una característica del PRTV es que el sistema no evalúa automáticamente cada expresión tan pronto como la recibe, sino que difiere la evaluación real hasta el último momento posible (vea la explicación sobre la formulación de consultas paso a paso en el capítulo 6, sección 6.5). Por lo tanto, la "única expresión relacional" del título del artículo podría representar de hecho una secuencia completa de operaciones del usuario. La optimización descrita se parece a la de SQUIRAL, pero va más allá en algunos aspectos e incluye lo siguiente (en orden de aplicación):

- Las restricciones son realizadas lo más pronto posible.
- Las secuencias de proyecciones son combinadas en una sola proyección.
- Las operaciones redundantes son eliminadas.
- Las expresiones que involucran relaciones vacías y condiciones triviales son simplificadas.
- Las subexpresiones comunes son factorizadas.

El artículo concluye con algunos resultados experimentales y algunas sugerencias para investigaciones adicionales.

17.20 Matthias Jarke y Jürgen Koch: "Range Nesting: A Fast Method to Evaluate Quantified Queries", Proc. 1983 ACM SIGMOD Int. Conf. on Management of Data, San José, Calif, (mayo, 1983).

Define una variación del cálculo relacional que permite aplicar algunas reglas de transformación sintáctica adicionales (y útiles) y presenta algoritmos para la evaluación de expresiones de ese cálculo. (De hecho, el cálculo particular que se describe está muy cercano al cálculo de tupias tal como lo describo en el capítulo 7.) El artículo describe la optimización de una clase particular de expresiones del cálculo revisado llamadas "expresiones anidadas perfectas". Proporciona métodos para convertir consultas aparentemente complejas —en particular determinadas consultas que involucran a FORALL— en expresiones perfectas. Los autores muestran que un gran subconjunto de las consultas que se presentan en la práctica, corresponden a expresiones perfectas.

17.21 Surajit Chaudhuri y Kyuseok Shim: "Including Group-By in Query Optimization", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

17.22 A. Makinouchi, M. Tezuka, H. Kitakami y S. Adachi: "The Optimization Strategy for Query Eva luation in RDB/V1", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, Francia (septiembre, 1981).

El RDB/V1 fue el prototipo a la cabeza del producto de Fujitsu AIM/RDB (que es un sistema SQL). Este artículo describe las técnicas de optimización usadas en ese prototipo y las compara brevemente con las usadas en los prototipos de Ingres y System R. Una técnica particular parece ser nueva: el uso de valores MAX y MIN obtenidos dinámicamente para inducir restricciones adicionales. Esta técnica tiene el efecto de simplificar el proceso de selección de un orden de junta y mejorar el desempeño de las propias juntas. Como un ejemplo simple de esto último, supongamos que los proveedores y las partes se van a juntar sobre las ciudades. Primero, los proveedores se ordenan por CIUDAD y durante el ordenamiento se determinan los valores máximo y mínimo (digamos MAX y MIN) de V.CIUDAD. Entonces se puede usar la restricción

```
MIN < P. CIUDAD AND P. CIUDAD < MAX
```

para reducir la cantidad de partes que necesitan ser inspeccionadas en la construcción de la junta.

17.23 Hamid Pirahesh, Joseph M. Hellerstein y Waqar Hasan: "Extensible Rule Based Query Rewrite Optimization in Starburst", Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, Calif, (junio, 1992).

Como indiqué en la sección 17.1, la "reescritura de consultas" es otro nombre para la transformación de expresiones. Los autores dicen que, sorprendentemente, los productos comerciales hacen muy poco con respecto a esas transformaciones (al menos en 1992). Sea como fuere, el artículo describe el mecanismo de transformación de expresiones en el prototipo Starburst de IBM (vea las referencias [17.50], [25.14], [25.17] y [25.21] a [25.22]). Los usuarios calificados adecuadamente pueden en cualquier momento añadir nuevas reglas de transformación al sistema (y a eso de debe el término "extensible" en el título del artículo).

17.24 Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh y Raghu Ramakrishnan: "Magic is Relevant", Proa 1990 ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, N.J. (mayo, 1990).

El inapropiado término "mágico" se refiere a una técnica de optimizacion desarrollada originalmente para ser usada con consultas —en especial consultas que involucran recursion— expresadas en el lenguaje "de base de datos lógica" Datalog (vea el capítulo 23). El presente artículo extiende el enfoque a los sistemas relaciónales convencionales, diciendo que, con base en mediciones experimentales, esta técnica es a menudo más efectiva que las técnicas de optimizacion tradicionales (observe que la consulta no tiene que ser recursiva para que este enfoque sea aplicable). La idea básica es descomponer la consulta dada en varias consultas más pequeñas que definen un conjunto de "relaciones auxiliares" (algo parecido al enfoque de descomposición de consulta que explicamos en la sección 17.6) como un medio para filtrar tupias que son irrelevantes para el problema que se tiene. El siguiente ejemplo (expresado en cálculo relacional) está basado en uno que da el artículo. La consulta original es:

```
R := EX.ENOMBRE WHERE EX.TRABAJO = 'Empleado' AND EX.SALARIO >
               AVG ( EY WHERE EY.DEPTO# - EX.DEPT0#, SALARIO ) ;
```

("Obtener el nombre de los empleados —es decir, trabajadores cuya categoría es empleado— cuyo salario sea mayor que el promedio de su departamento".) Si esta consulta se ejecuta "directamente" —es decir, más o menos como está escrita— el sistema revisará a los empleados tupia por tupia y por lo tanto, calculará varias veces el salario promedio de cualquier departamento que emplee a más de un empleado. Un optimizador tradicional podría por lo tanto dividir la consulta en las dos consultas más pequeñas siguientes:

```
T1 := ( EX.DEPT0#,
        AVG ( EY WHERE EY.DEPTO# ■ EX.DEPT0#, SALARIO ) AS ASAL ) ;

T2 := EMP.ENOMBRE WHERE EMP.TRABAJO = 'Empleado' AND
                  EXISTS T1 ( EMP.DEPT0# = T1.DEPTO# AND
                              EMP.SALARIO > T1.ASAL ) ;
```

Ahora ningún promedio de departamento se calculará más de una vez, pero se calcularán algunos promedios irrelevantes (en particular los de los departamentos que no tienen empleados).

El enfoque "mágico" evita los cálculos repetidos del primer enfoque y los cálculos irrelevantes del segundo; con el costo de que genera relaciones "auxiliares" adicionales:

```
/* primera relación auxiliar: nombre, departamento y salario    */
/* de los empleados                                              */
T1   :=   ( EMP.ENOMBRE, EMP.DEPTO*, EMP.SALARIO )
                                    WHERE EMP.TRABAJO = 'Empleado' ;

/* segunda relación auxiliar: departamentos que tienen empleados */
T2  :=  T1.DEPT0* ;

/* tercera relación auxiliar: departamentos que tienen empleados */
/* y sus correspondientes salarios promedio                      */
T3  :=  ( T2.DEPT0#,
            AVG ( EMP WHERE EMP.DEPTO# = T2.DEPT0*, SALARIO )
                                                    AS ASAL ) ;
/* relación resultante */
R   :=  T1.ENOMBRE WHERE EXISTS T3 ( T1.DEPT0# = T3.DEPT0# AND
                                     T1.SALARIO > T3.ASAL ) ;
```

Lo "mágico" consiste en determinar exactamente qué relaciones auxiliares se necesitan.

Para conocer otras referencias a "mágico" vea las referencias [17.25] y [17.26] que vienen a continuación y en la sección de "Referencias y bibliografía" del capítulo 23.

17.25 Inderpal Singh Mumick y Hamid Pirahesh: "Implementation of Magic in Starburst", Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, Minn, (mayo, 1994).

17.26 Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh y Raghu Ramakrishnan: "Magic Conditions", *ACM TODS 21,* No. 1 (marzo, 1996).

17.27 Jonathan J. King: "QUIST: A System for Semantic Query Optimization in Relational Data bases", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, Francia (septiembre, 1981).

Es el artículo que presentó la idea de la optimización semántica (vea la sección 17.4 de este capítulo). Describe un sistema experimental llamado QUIST ("mejora de consultas mediante transformación semántica") que es capaz de realizar tales optimizaciones.

17.28 Sreekumar T. Shenoy y Z. Meral Ozsoyoglu: "A System for Semantic Query Optimization", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo-junio, 1987).

Extiende el trabajo de King [17.27] al presentar un esquema que selecciona dinámicamente —entre un conjunto muy grande de restricciones de integridad— sólo a las restricciones que podrían ser rentables en la transformación de una consulta dada. Las restricciones de integridad consideradas son de dos tipos básicos, *restricciones de implicación* y *restricciones de subconjunto.* Un ejemplo de una restricción de implicación es "si la cantidad del embarque es mayor de 300, la ciudad del proveedor debe ser Londres" y un ejemplo de una restricción de subconjunto es "el conjunto de proveedores de Londres debe ser un subconjunto de los proveedores de envíos" (es decir, todo proveedor de Londres debe proporcionar al menos una parte). Tales restricciones son usadas para la transformación de consultas, eliminando restricciones y juntas redundantes e introduciendo restricciones adicionales sobre atributos indexados. Los casos donde las consultas pueden ser respondidas sólo con las restricciones, también son tratados eficientemente.

17.29 Michael Siegel, Edward Sciore y Sharon Salveter: "A Method for Automatic Rule Derivation to Support Semantic Query Optimization", *ACM TODS 17,* No. 4 (diciembre, 1992).

Como expliqué en la sección 17.4, la optimización semántica utiliza las restricciones de integridad para la transformación de consultas. Sin embargo, existen varios problemas asociados con esta idea:

» ¿Cómo sabe el optimizador qué transformaciones serán más efectivas (es decir, harán que la consulta sea más eficiente)?

• Algunas restricciones de integridad no son muy útiles para efectos de optimización. Por ejemplo, el hecho de que el peso de las partes deba ser mayor que cero, aunque es importante para efectos de integridad, es esencialmente inútil para la optimización. ¿Cómo distingue el optimizador entre las restricciones útiles y las inútiles?

■ Algunas condiciones pueden ser válidas para algunos estados de la base de datos —e incluso para la mayoría de los estados— y por lo tanto ser útiles para los propósitos de la optimización, sin ser estrictamente restricciones de integridad como tales. Un ejemplo podría ser la condición "la edad del empleado es menor o igual a 50"; aunque no es una restricción de integridad como tal (los empleados pueden ser mayores de 50 años), bien puede darse el caso que ningún empleado actual sea de hecho mayor de 50 años.

Este artículo describe la arquitectura de un sistema que trata los asuntos anteriores.

17.30 Upen S. Chakravarthy, John Grant y Jack Minker: "Logic Based Approach to Semantic Query Optimization", *ACM TODS 15,* No. 2 (junio, 1990).

Para citar el resumen: "en varios artículos anteriores [los autores] han descrito y probado la corrección de un método para optimizar semánticamente las consultas... Este artículo consolida los principales resultados de esos artículos, enf atizando las técnicas y su aplicabilidad para la optimi-

zación de consultas relaciónales. Además [muestra], la manera en que este método incluye y generaliza trabajos anteriores sobre la optimización semántica de consultas. [También indica] la manera en que pueden ser ampliadas las técnicas de optimización semántica de consultas hacia [consultas recursivas] y restricciones de integridad que contienen disyunción, negación y recursion".

17.31 A. V. Aho, Y. Sagiv y J. D. Ullman: "Efficient Optimization of a Class of Relational Expressions", *ACM TODS 4*, No. 4 (diciembre, 1979).

La clase de expresiones relaciónales a la que el título de este artículo hace referencia, es aquella que involucra solamente restricciones de igualdad (mencionadas en el texto como *selecciones),* proyecciones y juntas naturales (llamadas *expresiones SPJ).* Las expresiones SPJ corresponden a las consultas del cálculo relacional que involucran solamente comparaciones de igualdad, AND y cuantificadores existenciales. El artículo presenta a las tableaux como un medio de representación simbólica de las expresiones SPJ. Una tableau es un arreglo rectangular donde las columnas corresponden a los atributos y las filas a las condiciones; específicamente a las *condiciones de pertenencia,* las cuales establecen que exista una determinada (sub)tupla en una relación determinada. Las filas están conectadas en forma lógica por la aparición de símbolos comunes en las filas afectadas. Por ejemplo, la tabla

V#	STATUS	CIUDAD	P#	COLOR
	a1			
b1	*a1*	Londres	---	proveedores
bi		*b2*	---	envíos
		b2 Rojo	---	partes

representa la consulta "obtener el status *(al)* de los proveedores *(bl)* en Londres que proporcionan alguna parte roja *(b2)".* La fila superior de la tableau, lista todos los atributos mencionados en la consulta. La siguiente fila es la fila de "resumen" (que corresponde a la prototupla en una consulta de cálculo o a la proyección final en una consulta algebraica) y las filas restantes (como ya se dijo) representan las condiciones de pertenencia. Hemos etiquetado esas filas en el ejemplo para que indiquen las relaciones relevantes (o varrels, más bien). Observe que las *"b"* se refieren a variables ligadas y las "a" a las variables libres; la fila de resumen sólo contiene *"a".*

Las tableaus representan otro candidato para un formalismo canónico de las consultas (vea la sección 17.3), con excepción de que (por supuesto) no son lo suficientemente generales para representar todas las expresiones relaciónales posibles. (De hecho, pueden ser consideradas como una variación sintáctica de la QBE —consulta-por-ejemplo— que es sin embargo estrictamente menos poderosa que esta última.) El artículo da algoritmos para reducir cualquier tableau en otra equivalente semánticamente, en la cual la cantidad de filas es reducida a un límite. Puesto que la cantidad de filas (sin contar a las dos primeras que son especiales) es de una más que la cantidad de juntas en la expresión SPJ correspondiente, la tableau convertida representa una forma óptima de la consulta; óptima en el sentido muy específico de que se minimiza la cantidad de juntas. (Por supuesto, en el ejemplo anterior la cantidad de juntas ya es el mínimo posible para la consulta y tal optimización no sucede.) La tableau mínima puede luego convertirse —si se desea— en alguna otra representación para una optimización adicional subsecuente.

La idea de minimizar la cantidad de juntas tiene aplicabilidad en las consultas formuladas en términos de vistas de junta (en particular, las consultas formuladas en términos de una "relación universal"; vea la sección de "Referencias y bibliografía" en el capítulo 12). Por ejemplo, supongamos que al usuario se le presenta una vista VE que está definida como la junta de proveedores y envíos sobre V# y el usuario emite la consulta:

```
VE { P# }
```

Un algoritmo de procesamiento de vistas directo convertiría esta consulta en lo siguiente:

```
( VP JOIN V ) { P# }
```

Sin embargo, como señalé en la sección 17.4, la siguiente consulta produce el mismo resultado y no involucra una junta (es decir, la cantidad de juntas se ha minimizado):

```
VP { p# }
```

Por lo tanto, observe que ya que los algoritmos para la reducción de tableau dados en el artículo, toman en cuenta cualquier dependencia funcional establecida explícitamente entre los atributos (vea el capítulo 10), esos algoritmos proporcionan un ejemplo limitado de una técnica de optimización *semántica.*

17.32 Y. Sagiv y M. Yannakakis: "Equivalences Among Relational Expressions with the Union and Difference Operators", *J'ACM 27,* No. 4 (octubre, 1980).

Amplía las ideas de la referencia [17.31] para incluir consultas que utilizan operaciones de unión y diferencia.

17.33 Alón Y. Levy, Inderpal Singh Mumick y Yehoshua Sagiv: "Query Optimization by Predicate Move-Around", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

17.34 P. Griffiths Selinger *et al.*\ "Access Path Selection in a Relational Database System", Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass, (mayo-junio, 1979).

Este importante artículo explica algunas de las técnicas de optimización usadas en el prototipo del System R. *Nota:* El optimizador del System R fue el antecesor del optimizador del DB2. La referencia [17.35] da información adicional específica del DB2.

Una consulta en System R es una instrucción SQL y por lo tanto, consiste en un conjunto de bloques "SELECT-FROM-WHERE" *{bloques de consulta),* algunos de los cuales pueden estar anidados dentro de otros. El optimizador del System R selecciona primero un orden en el cual ejecutar esos bloques de consulta y luego busca minimizar el costo total de la consulta seleccionando la implementación más barata para cada bloque individual. Observe que esta estrategia (primero seleccionar el orden de los bloques y luego optimizar bloques individuales) significa que algunos planes de consulta determinados nunca serán considerados, ya que de hecho usa una técnica para "reducir el espacio de búsqueda" (vea los comentarios sobre este tema cerca del final de la sección 17.3). *Nota:* En el caso de los bloques anidados, el optimizador sigue simplemente el orden de anidamiento tal como lo especifica el usuario; es decir, el bloque más interno será ejecutado primero, en términos generales. Vea las referencias [17.39] a [17.45] para críticas y explicaciones adicionales sobre esta estrategia.

Para un bloque de consulta dado existen básicamente dos casos a considerar (de hecho, el primero puede ser considerado como un caso especial del segundo):

1. Para un bloque que involucra solamente una restricción o proyección de una sola relación, el optimizador usa información estadística del catálogo, junto con fórmulas (dadas en el artículo) para estimar el tamaño de los resultados intermedios y el costo de las operaciones de bajo nivel, con el fin de escoger una estrategia para la realización de esa restricción o proyección
2. Para un bloque que involucra dos o más relaciones a ser juntadas, con (probablemente) res tricciones o proyecciones locales, el optimizador (a) trata cada relación individual como en el caso 1 y (b) selecciona una secuencia para la realización de las juntas. Las dos operaciones (a) y (b) no son independientes; por ejemplo, una estrategia dada —digamos, el uso de un

determinado índice— para el acceso a una relación individual *A,* bien puede ser seleccionada debido precisamente a que produce tupias de *A* en el orden necesario para realizar una junta subsecuente de *A* con alguna otra relación *B.*

Las juntas son implementadas por ordenamiento/mezcla, búsqueda con índice o fuerza bruta. Un punto que se enfatiza en el artículo es que en la evaluación de, por ejemplo, la junta anidada *(A* JOIN *B)* JOIN C, no es necesario calcular completamente la junta de *A* y *B* antes de calcular la junta del resultado y C; por el contrario, tan pronto como se ha producido una tupia de *A* JOIN *B* puede ser pasada de inmediato al proceso que junta esas tupias con las tupias de *C.* Por lo tanto, probablemente nunca será necesario materializar completamente la relación "A JOIN *B".* (Esta idea general de *canalización* la explicamos brevemente en el capítulo 3 sección 3.2. Vea también las referencias [17.18] y [17.60].)

Este artículo también incluye algunas observaciones sobre el costo de la optimización. Para una junta de dos relaciones, decimos que el costo es aproximadamente igual al de entre 5 y 20 recuperaciones de la base de datos; una sobrecarga insignificante si la consulta optimizada será ejecutada subsecuentemente muchas veces. (Observe que el System R es un sistema de compilación y por lo tanto, una instrucción SQL puede ser optimizada una vez y luego ejecutada muchas veces, tal vez muchas miles de veces.) Decimos que la optimización de consultas complejas requiere "sólo unos cuantos miles de bytes de almacenamiento y unas cuantas décimas de segundo" en un IBM System 370 modelo 158. "Las juntas de ocho tablas han sido optimizadas en unos cuantos minutos."

17.35 J. M. Cheng, C. R. Loosley, A. Shibamiya y P. S. Worthington: "IBM DATABASE 2 Perfor mance: Design, Implementation, and Tuning", *IBM Sys. J. 23,* No. 2 (1984).

Incluye una breve descripción de las tácticas de optimización en DB2 (en su primera versión): técnicas de transformación de consultas, el manejo de bloques de consulta anidados, métodos de junta, selección de ruta de acceso y procesamiento sólo con índices. *Nota:* El artículo también incluye material muy interesante relacionado con otros aspectos orientados al desempeño del DB2.

17.36 Eugene Wong y Karel Youssefi: "Decomposition—A Strategy for Query Processing", *ACM TODS I,* No. 3 (septiembre, 1976).

17.37 Karel Youssefi y Eugene Wong: "Query Processing in a Relational Database Management System", Proc. 5th Int. Conf. on Very Large Data Bases, Río de Janeiro, Brasil (septiembre, 1979).

17.38 Lawrence A. Rowe y Michael Stonebraker: "The Commercial Ingres Epilogue", en la referen cia [7.10].

"Ingres comercial" es el producto que surgió a partir del prototipo de "Ingres Universitario". Algunas de las diferencias entre los optimizadores del Ingres universitario y el comercial son las siguientes:

1. El optimizador universitario usa "planeación incremental"; es decir, decide primero qué hacer, lo hace, decide lo que hay que hacer después con base en el tamaño del resultado del paso an terior y así sucesivamente. El optimizador comercial decide un plan completo antes de comen zar la ejecución, basándose en estimaciones de los tamaños de los resultados intermedios.
2. El optimizador universitario maneja consultas de dos variables (es decir, junta) por sustitu ción de tupia, como explico en la sección 17.6. El optimizador comercial soporta una variedad de técnicas preferidas para el manejo de dichas consultas, incluyendo en particular la técnica de ordenamiento/mezcla descrita en la sección 17.7.
3. El optimizador comercial usa un conjunto de estadísticas mucho más sofisticadas que el opti mizador universitario.
4. El optimizador universitario hace planeación incremental (como mencioné en el punto 1). El optimizador comercial hace una búsqueda más exhaustiva. Sin embargo, el proceso de búsqueda

se detiene cuando el tiempo gastado en la optimización excede la mejor estimación actual del tiempo requerido para ejecutar la consulta (porque de no hacerlo así, la sobrecarga de la optimización bien puede sobrepasar las ventajas).

5. El optimizador comercial considera todas las combinaciones de índices posibles, todas las secuencias de junta posibles y "todos los métodos de junta disponibles: ordenamiento/mezcla, ordenamiento/mezcla parcial, búsqueda con dispersión, búsqueda ISAM, búsqueda en árbol B y fuerza bruta" (vea la sección 17.7).

17.39 Won Kim: "On Optimizing an SQL-Like Nested Query", *ACM TODS* 7, No. 3 (septiembre, 1982).

Vea más adelante el comentario a la referencia [17.43].

17.40 Werner Kiessling: "On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates", Proc. 1 lth Int. Conf. on Very Large Data Bases, Estocolmo, Suecia (agosto, 1985).

Vea más adelante el comentario a la referencia [17.43].

17.41 Richard A. Ganski y Harry K. T. Wong: "Optimization of Nested SQL Queries Revisited", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

Vea más adelante el comentario a la referencia [17.43].

17.42 Günter von Bültzingsloewen: "Translating and Optimizing SQL Queries Having Aggregates", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

Vea más adelante el comentario a la referencia [17.43].

17.43 M. Muralikrishna: "Improved Unnesting Algorithms for Join Aggregate SQL Queries", Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (agosto, 1992).

En términos generales, el lenguaje SQL permite "subconsultas anidadas", es decir, un bloque SELECT-FROM-WHERE que está anidado dentro de otro bloque similar (vea el capítulo 7). Esta construcción ha causado grandes problemas a los implementadores. Considere la siguiente consulta SQL ("obtener los nombres de los proveedores que proporcionan la parte P2"), a la cual nos referiremos como consulta Ql:

```
SELECT V.PROVEEDOR
FROM   V WHERE  V.V#
IN
     ( SELECT VP.V# FROM VP
       WHERE VP.P# = 'P2' ) ;
```

En el System R [17.34] esta consulta será implementada evaluando primero el bloque interno para producir una tabla temporal, digamos T, que contenga los *números* de proveedor para los proveedores requeridos, luego será revisada la tabla V —fila por fila— y para cada fila se revisará la tabla T para ver si contiene el número de proveedor correspondiente. Es probable que esta estrategia sea bastante ineficiente (en especial si la tabla T no se indexa). Ahora considere la siguiente consulta (Q2):

```
SELECT V.PROVEEDOR
FROM   V, VP
WHERE  V.V# = VP.V#
AND    VP.P# = 'P2' ;
```

Vemos claramente que esta consulta es semánticamente idéntica a la anterior, pero el System R ahora considerará estrategias adicionales de implementación para ella. En particular, si las tablas V y VP están almacenadas físicamente en secuencia de número de proveedor, usará una junta

con mezcla que será muy eficiente. Y tomando en cuenta que (a) las dos consultas son lógicamente equivalentes pero, (b) la segunda es inmediatamente más susceptible para una implementación eficiente, parece que vale la pena explorar la posibilidad de transformar las consultas de tipo Ql hacia consultas de tipo Q2. Esta posibilidad es el tema de las referencias [17.39] a [17.45],

Kim [17.39] fue el primero en tratar este problema. Identificó cinco tipos de consultas anidadas y describió los algoritmos de transformación correspondientes. El artículo de Kim incluye algunas mediciones experimentales que muestran que los algoritmos propuestos mejoran el desempeño de las consultas anidadas en uno o dos órdenes de magnitud (en general).

Posteriormente, Kiessling [17.40] mostró que los algoritmos de Kim no funcionaban correctamente si una subconsulta anidada (a cualquier nivel) incluía un operador COUNT en su lista SELECT (no manejaba adecuadamente el caso en donde el argumento de COUNT daba como resultado un conjunto vacío). Los "escollos semánticos" del título del artículo se refieren a las dificultades y complejidades del SQL por las que los usuarios tienen que navegar para obtener respuestas consistentes y correctas a dichas preguntas. Además, Kiessling también mostró que el algoritmo de Kim no era fácil de arreglar ("parece que no hay una manera uniforme para hacer estas transformaciones en forma eficiente y correcta bajo todas las circunstancias").

El artículo de Ganski y Wong [17.41] proporciona una solución al problema identificado por Kiessling mediante el uso de una *junta externa* (vea el capítulo 18), en lugar de la junta interna normal en la versión transformada de la consulta. (En mi opinión, la solución no es totalmente satisfactoria, debido a que presenta una dependencia de ordenamiento indeseable entre los operadores en la consulta transformada.) El artículo también identifica un error adicional en el artículo original de Kim y lo resuelve de la misma forma. Sin embargo, las transformaciones de este artículo contienen errores adicionales propios; algunos tienen que ver con el problema de filas duplicadas (un notorio "escollo semántico" [17.40]) y otros con el comportamiento defectuoso del cuantificador EXISTS del SQL [18.6],

El artículo de Von Bültzingsloewen [17.42] representa un intento para poner todo el tema sobre unas bases teóricas firmes (donde el problema básico es que —como han observado varios escritores— el comportamiento tanto sintáctico como semántico del anidamiento y la agregación al estilo SQL, no está bien comprendido). Define versiones extendidas del cálculo relacional y el álgebra relacional (que tienen que ver con las extensiones con totales y nulos), y prueba la equivalencia de estos dos formalismos extendidos (usando de paso un nuevo método de prueba que parece más elegante que los publicados anteriormente). Luego define la semántica del SQL transformando el SQL hacia el cálculo extendido definido. Sin embargo, debe observarse que:

1. El dialecto de SQL que se explica, aunque es cercano al dialecto típico soportado en los pro ductos comerciales explicados en las referencias [17.39] a [17.41], todavía no es completa mente ortodoxo; no incluye UNION, no soporta directamente a los operadores de la forma "=ALL" o ">ALL" (vea el apéndice A) y su tratamiento de los valores de verdad *des conocidos* es diferente (y de hecho mejor) que el que hace el SQL convencional.
2. El artículo omite consideraciones sobre asuntos que tienen que ver con la eliminación de du plicados "por simplificación técnica". Pero las implicaciones de esta omisión no quedan claras, tomando en cuenta que (como dije anteriormente) la posibilidad de duplicados tiene consecuencias importantes para la validez de determinadas transformaciones [5.6].

Por último, Muralikrishna [17.43] dice que el algoritmo original de Kim [17.39], aunque incorrecto, todavía puede ser más eficiente que la "estrategia general" de la referencia [17.41] en algunos casos y, por lo tanto, propone una corrección alterna al algoritmo de Kim. También proporciona algunas mejoras adicionales.

17.44 Lars Baekgaard y Leo Mark: "Incremental Computation of Nested Relational Query Expressions", *ACM TODS 20,* No. 2 (junio, 1995).

Es otro artículo sobre la optimización de consultas que involucran subconsultas al estilo SQL, en especial las *correlacionadas* (el "anidamiento" mencionado en el título del artículo se refiere específicamente a las subconsultas anidadas estilo SQL). La estrategia es (1) convertir la consulta original en una equivalente no anidada y luego (2) evaluar en forma incremental la versión no anidada. "Para apoyar el paso (1) hemos desarrollado un algoritmo de transformación de álgebra-álgebra muy conciso... La expresión [transformada] hace uso intensivo del operador [MINUS]. Para apoyar el paso (2) presentamos y analizamos un algoritmo eficiente para la evaluación incremental [operaciones MINUS]". El término *cálculo incremental* se refiere a la idea de que la evaluación de una consulta dada puede utilizar los resultados calculados anteriormente.

17.45 Jun Rao y Kenneth A. Ross: "Using Invariants: A New Strategy for Correlated Queries", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

Otro artículo sobre la optimización de consultas que involucran subconsultas al estilo SQL.

17.46 David H. D. Warren: "Efficient Processing of Interactive Relational Database Queries Ex pressed in Logic", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, Francia (septiembre, 1981).

Presenta una vista de la optimización de consultas desde una perspectiva bastante diferente: la de la lógica formal. El artículo reporta las técnicas usadas en un sistema de base de datos experimental basado en Prolog. Las técnicas son aparentemente muy similares a las del System R, aunque se llegó a ellas en forma bastante diferente y con objetivos algo diferentes. El artículo sugiere que al contrario de lo que sucede con los lenguajes de consulta convencionales como QUEL y SQL, los lenguajes basados en la lógica como Prolog permiten consultas expresadas en forma tal que hagan resaltar:

- Cuáles son los componentes esenciales de la consulta, específicamente las metas lógicas.
- Qué es los que vincula a estos componentes, específicamente las variables lógicas.
- Cuál es el problema de implementación crucial, específicamente la secuencia en la cual se tratan de satisfacer las metas.

Por consecuencia, se sugiere que un lenguaje de éstos es muy conveniente como una base para la optimización. Además, podría ser considerado como otro candidato para la representación interna de las consultas que son expresadas originalmente en otro lenguaje (vea la sección 17.3).

17.47 Yannis E. Ioannidis y Eugene Wong: "Query Optimization by Simulated Annealing", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

La cantidad de planes de consulta posibles crece exponencialmente con la cantidad de relaciones involucradas en la consulta. En aplicaciones comerciales convencionales, la cantidad de relaciones en una consulta es generalmente pequeña y por lo tanto, la cantidad de planes candidatos (el "espacio de búsqueda") permanece por lo general dentro de límites razonables. Sin embargo, en aplicaciones más recientes, la cantidad de relaciones en una consulta puede fácilmente llegar a ser muy grande (vea por ejemplo el capítulo 21). Además, también es probable que esas aplicaciones necesiten optimización "global" (es decir, consultas múltiples) [17.49] y soporte de consultas recursivas, donde ambas tienen el potencial de incrementar significativamente el espacio de búsqueda. La búsqueda exhaustiva se está haciendo rápidamente imposible en tales ambientes y comienza a ser imperativa alguna técnica eficaz para reducir el espacio de búsqueda.

El presente artículo da referencias de trabajos anteriores sobre los problemas de optimización para grandes cantidades de relaciones y optimización de consultas múltiples, pero dice que no han sido publicados algoritmos anteriores para la optimización de las consultas recursivas. Luego presenta un algoritmo que dice ser adecuado cuando el espacio de búsqueda es grande y en particular muestra la manera de aplicar ese algoritmo al caso de consultas recursivas. El algoritmo (llamado "templado simulado", debido a que simula el proceso de templado por medio del cual los cristales se desarrollan primero por medio del calentamiento del fluido que los contiene y luego permitiendo

que se enfríen gradualmente) es un algoritmo probabilístico del tipo "subir la cuesta" que ha sido aplicado satisfactoriamente para la optimizacion de problemas en otros contextos. Vea también la referencia [17.48] que viene a continuación.

17.48 Arun Swami y Anoop Gupta: "Optimization of Large Join Queries", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. (junio, 1988).

El problema general de la determinación del orden de junta óptimo en consultas que involucran gran cantidad de relaciones (por ejemplo, cómo se presenta con los sistemas de bases de datos deductivos, vea el capítulo 23) es en conjunto difícil. Este artículo presenta un análisis comparativo de varios algoritmos que atacan este problema: caminata de perturbación, muestreo cuasialeatorio, mejora iterativa, heurística de secuencia y templado simulado [17.47] (los nombres añaden un elemento poético a un tema que, de lo contrario, sería un tanto prosaico). De acuerdo con este análisis, la mejora iterativa es superior a todos los demás algoritmos; en particular, el templado simulado no es útil "por sí mismo" para las consultas de juntas grandes.

17.49 Timos K. **Seflis:** "Multiple-Query Optimization", *ACM TODS 13,* No. 1 (marzo, 1988).

La investigación de optimizacion clásica se ha enfocado en el problema de optimizar expresiones relaciónales aisladas. Sin embargo, en el futuro, la habilidad para optimizar varias consultas distintas como una unidad probablemente llegue a ser importante. Una razón para este estado de cosas es que lo que comienza como una sola consulta en algún nivel superior del sistema puede involucrar varias consultas a nivel relacional. Por ejemplo, la consulta en lenguaje natural "¿le pagan bien a Mike?" podría conducir posiblemente a la ejecución de tres consultas relaciónales independientes:

- "¿Mike gana más de $75,000?"
- "¿Mike gana más de $60,000 y tiene menos de cinco años de experiencia?"
- "¿Mike gana más de $45,000 y tiene menos de tres años de experiencia?"

Este ejemplo ilustra el hecho de que es probable que conjuntos de consultas relacionadas compartan algunas subexpresiones comunes y conduzcan por sí mismas, a una optimizacion global. El artículo considera consultas que involucran solamente conjunciones de restricciones o equijuntas. Incluye algunos resultados experimentales motivadores y da direcciones para una investigación futura.

17.50 Guy M. Lohman: "Grammar-Like Functional Rules for Representing Query Optimization Alternatives", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. (junio, 1988).

En cierto aspecto, un optimizador relacional puede ser considerado como un sistema experto, pero las reglas que manejan el proceso de optimizacion han sido incrustadas históricamente en código de procedimientos y no establecidas en forma separada y declarativa. Por consecuencia, extender el optimizador para que incorpore nuevas técnicas de optimizacion no ha sido fácil. Los sistemas de bases de datos futuros (vea el capítulo 25) aumentarán este problema, ya que habrá una clara necesidad de que las instalaciones individuales extiendan al optimizador para que incorpore (por ejemplo) soporte para tipos de datos específicos definidos por el usuario. Por lo tanto, varios investigadores han propuesto la estructuración del optimizador como un sistema experto convencional con reglas declarativas indicadas explícitamente.

Sin embargo, esta idea tiene ciertos problemas de desempeño. En particular, es posible aplicar una gran cantidad de reglas en cualquier etapa dada del procesamiento de la consulta y la determinación de la regla adecuada puede involucrar cálculos complejos. El presente artículo describe un enfoque alterno (que implementa el prototipo Starburst, vea las referencias [25.14], [25.17], [25.21] y [25.22]) donde las reglas están establecidas por medio de reglas de producción en una gramática parecida a la que se usa para describir los lenguajes formales. Las reglas, llamadas STARs (reglas de estrategia alterna) permiten la construcción recursiva de planes de

consulta a partir de otros planes y "operadores de planes de bajo nivel" (LOLEPOPs), que son operaciones básicas sobre relaciones tales como junta, ordenamiento, etcétera. Los LOLEPOPs vienen en varios *tipos* diferentes; por ejemplo, la junta LOLEPOP tiene un tipo de ordenamiento/mezcla, un tipo de dispersión, etcétera.

El artículo afirma que este enfoque tiene varias ventajas. Las reglas (STARs) son fácilmente comprensibles por quienes necesitan definir nuevas, el proceso para determinar qué regla aplicar en cualquier situación dada es más simple y más eficiente que el enfoque de sistema experto más tradicional, y satisface el objetivo de la extensibilidad.

17.51 Ryohei Nakano: "Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions", *ACM TODS 15*, No. 4 (diciembre, 1990).

Como expliqué en el capítulo 7 (sección 7.4), las consultas en un lenguaje basado en cálculo pueden ser implementadas (a) traduciendo la consulta en consideración a una expresión algebraica equivalente, luego (b) optimizando esa expresión algebraica y por último, (c) implementando esa expresión optimizada. En este artículo Nakano propone un esquema para la combinación de los pasos (a) y (b) en un solo paso, y por lo tanto, traduce directamente una expresión de cálculo dada en un equivalente algebraico *óptimo.* Se dice que este esquema es "más efectivo y más promisorio... ya que parece menos difícil optimizar expresiones algebraicas complicadas". El proceso de traducción utiliza ciertas transformaciones *heurísticas,* donde incorpora el conocimiento humano con relación a la equivalencia de determinadas expresiones algebraicas y de cálculo.

17.52 Kyu-Young Whang y Ravi Krishnamurthy: "Query Optimization in a Memory-Resident Domain Relational Calculus Database System", *ACM TODS 15,* No. 1 (marzo, 1990).

El aspecto más caro del procesamiento de consultas (en el ambiente de memoria principal supuesto por este artículo) es la evaluación de expresiones lógicas. La optimización en ese ambiente está orientada a minimizar la cantidad de tales evaluaciones.

17.53 Johann Christoph Freytag y Nathan Goodman: "On the Translation of Relational Queries into Iterative Programs", *ACM TODS 14,* No. 1 (marzo, 1989).

Presenta métodos para compilar directamente expresiones relaciónales en código ejecutable en lenguajes como C o Pascal. Observe que este enfoque difiere del tratado en el cuerpo de este capítulo, donde el optimizador combina efectivamente fragmentos de código preescritos (con parámetros) para construir el plan de consulta.

17.54 Kiyoshi Ono y Guy M. Lohman: "Measuring the Complexity of Join Enumeration in Query Optimization", Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, Australia (agosto, 1990).

Dado que la junta es básicamente una operación diádica, el optimizador tiene que dividir una junta que involucra *N* relaciones *(N* > 2) en una secuencia de juntas diádicas. La mayoría de los optimizadores hace esto en forma estrictamente anidada; esto significa que primero selecciona un par de relaciones a juntar, luego una tercera a partir del resultado de la junta de las dos primeras y así sucesivamente. En otras palabras una expresión tal como *A* JOIN *B* JOIN C JOIN *D* puede ser tratada como *((D* JOIN S) JOÍN Q JOIN *A,* pero nunca como *(A* JOIN *D)* JOIN *(B* JOIN Q. Además, los optimizadores tradicionales están generalmente diseñados para evitar, en la medida de lo posible, productos cartesianos. Estas tácticas pueden ser consideradas como formas de "reducir el espacio de búsqueda" (aunque por supuesto, todavía son necesarias algunas técnicas para la selección de la secuencia de juntas).

El presente artículo describe los aspectos relevantes del optimizador en el prototipo Starburst de IBM (vea las referencias [17.50], [25.14], [25.17], [25.21] y [25.22]). Dice que las tácticas anteriores pueden ser inadecuadas en determinadas situaciones y por lo tanto, lo que se necesita es un optimizador *adaptable* que pueda ser instruido para que use tácticas diferentes para consultas diferentes. *Nota:* A diferencia de los optimizadores comerciales típicos de hoy día, Starburst

es capaz de tratar una expresión de la forma *R.A = S.B + c* como una condición de "junta". También aplica el "cierre transitivo de predicado" (vea la sección 17.4).

17.55 Bennet Vanee y David Maier: "Rapid Bushy Join-Order Optimization with Cartesian Products", Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (junio, 1996).

Como mencioné en el comentario de la referencia anterior, los optimizadores tienden a "reducir el espacio de búsqueda" evitando (entre otras cosas) los planes que involucran productos cartesianos. Este artículo muestra que la búsqueda del espacio completo "es más posible de lo que se ha reconocido anteriormente" y la evitación de los productos cartesianos no necesariamente es benéfica. De acuerdo con los autores, las contribuciones principales del artículo son (a) separar completamente la enumeración del orden de las juntas y el análisis de predicados, y (b) presentar "nuevas técnicas de implementation" para tratar el problema de la enumeración del orden de las juntas.

17.56 Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim y Timos K. Sellis: "Parametric Query Optimization", Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canada (agosto, 1992).

Considere la siguiente consulta:

```
EMP WHERE SALARIO > salario
```

(donde *salario* es un parámetro en tiempo de ejecución). Supongamos que hay un índice sobre SALARIO. Entonces:

- Si *salario* es $10,000 mensuales, la mejor forma para implementar la consulta es usar el índice (ya que probablemente no calificarán la mayoría de los empleados).
- Si *salario* es $1,000 mensuales, la mejor forma para implementar la consulta es con una re visión secuencial (ya que probablemente la mayoría de los empleados *sí* calificarán).

Este ejemplo ilustra el punto de que algunas decisiones de optimization se toman mejor en tiempo de ejecución, incluso en un sistema compilado. El presente artículo explora la posibilidad de la generación de *conjuntos* de planes de consulta en tiempo de compilación (en donde cada plan es "óptimo" para algún subconjunto del conjunto de todos los valores posibles de los parámetros en tiempo de ejecución) y luego la selección del plan adecuado en tiempo de ejecución cuando ya se conocen los valores de los parámetros reales. En particular se enfoca en un parámetro específico: la cantidad de espacio de buffer disponible para la consulta. Resultados experimentales muestran que el enfoque del artículo impone muy poca sobrecarga de tiempo en el proceso de optimización y sacrifica muy poco en términos de la calidad y de los planes generados; de acuerdo con ello se dice que el enfoque puede mejorar significativamente el desempeño de la consulta. "Los ahorros en el costo de ejecución por el uso de un plan que está hecho específicamente para los valores reales de los parámetros... pueden ser enormes".

17.57 Navin Kabra y David J. DeWitt: "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

17.58 Jim Gray: "Parallel Database Systems 101", Proc. 1995 ACM SIGMOD Int. Conf. on Mana gement of Data, San Jose, Calif, (mayo, 1995).

Éste no es un artículo de investigación sino un resumen amplio para una presentación tutorial. En general, la idea básica que está detrás de los sistemas paralelos, es dividir un problema grande en muchos pequeños que pueden ser resueltos simultáneamente (y de esta forma mejorar el desempeño). En particular, los sistemas de bases de datos relaciónales son muy manejables en cuanto a la paralelización, debido a la naturaleza del modelo relational; es conceptualmente fácil (a) dividir relaciones en subrelaciones de muchas formas y (b) dividir expresiones relaciónales en subexpresiones, de nuevo, en muchas formas. De acuerdo con el título de esta referencia, proporcionamos algunas palabras sobre determinados conceptos principales de los sistemas de base de datos paralelos.

En primer lugar, la arquitectura del hardware subyacente involucrará (presuntamente) por sí misma algún tipo de paralelismo. Hay tres arquitecturas principales y cada una de ellas involucra varias unidades de procesamiento, varias unidades de disco y una red de interconexión de algún tipo:

- *Memoria compartida:* La red permite que todos los procesadores accedan a la misma memoria.
- *Disco compartido:* Cada procesador tiene su propia memoria, pero la red permite que todos los procesadores accedan a todos los discos.
- *Nada compartido:* Cada procesador tiene su propia memoria y discos, pero la red permite que los procesadores se comuniquen entre sí.

En la práctica la arquitectura a escoger es por lo general "nada compartido", al menos para los sistemas grandes (los otros dos enfoques caen rápidamente en problemas de *interferencia* conforme se añaden más procesadores). Para ser específicos, compartir nada proporciona una **aceleración lineal** (el incremento de hardware en un factor *N* mejora el tiempo de respuesta en un factor AO y una **escalación lineal** (el incremento del hardware y el volumen de datos por el mismo factor mantiene constante el tiempo de respuesta). *Nota:* A la "escalación" también se le conoce como **escalabilidad.**

También existen varios enfoques para la *partición de datos* (es decir, la división de una relación r en particiones o subrelaciones y la asignación de esas particiones a *n* procesadores diferentes):

- *Partición por rango:* La relación *r* se divide en particiones disjuntas 1, 2, ..., *n* con base en los valores de algún subconjunto *s* de los atributos de r (r está ordenada conceptualmente bajo i y el resultado se divide en n particiones de igual tamaño). Luego se asigna la partición *i* al procesador *i.* Este enfoque es bueno para consultas que involucran restricciones de igualdad o rango sobre *s.*
- *Partición por dispersión:* A cada tupia *t* de *r* se le asigna un procesador *h(t),* donde *h* es alguna función de dispersión. Este enfoque es bueno para consultas que involucran una restricción de igualdad sobre los atributos de dispersión y también es buena para consultas que involu cran acceso secuencial a toda la relación *r.*
- *Partición en ronda: r* está conceptualmente ordenado de alguna forma; a la tupia ¡esima del resultado ordenado le es asignado el procesador *i* módulo *n.* Este enfoque es bueno para con sultas que involucran el acceso secuencial a toda la relación r.

El paralelismo puede aplicarse a la ejecución de una operación individual (paralelismo *intraoperaciór),* a la ejecución de distintas operaciones dentro de la misma consulta (paralelismo *interoperación* o *intraconsulta)* y a la ejecución de distintas consultas (paralelismo *interconsulta).* Las referencias [17.4] y [17.61] contienen tutoriales sobre todas estas posibilidades, y las referencias [17.59] y [17.60] tratan algunas técnicas y algoritmos específicos. Hacemos notar que una versión paralela de la *junta por dispersión* (vea la sección 17.7) es particularmente efectiva y ampliamente usada en la práctica.

17.59 Dina Bitton, Harán Boral, David J. DeWitt y W. Kevin Wilkinson: "Parallel Algorithms for the Execution of Relational Database Operations", *ACM TODS 8,* No. 3 (septiembre, 1983).

Presenta algoritmos para la implementación de operaciones de ordenamiento, proyección, junta, agregación y actualización en un ambiente de varios procesadores. El artículo da fórmulas generales de costo que toman en cuenta los costos de E/S, mensajes y procesador, y que pueden ajustarse para diferentes arquitecturas de varios procesadores.

17.60 Waqar Hasan y Rajeev Motwani: "Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

17.61 Abraham Silberschatz, Henry F. Korth y S. Sudarshan: *Database System Concepts* (3a edi ción). Nueva York, N.Y.: McGraw-Hill (1997).

Este libro de texto general sobre administración de base de datos incluye un capítulo completo sobre sistemas de base de datos paralelos, así como uno sobre "arquitecturas de sistemas de base de datos" (centralizadas frente a cliente-servidor, paralelas y distribuidas).

RESPUESTAS A EJERCICIOS SELECCIONADOS

17.1 a. Válido, b. Válido, c. Válido, d. No válido, e. Válido, f. No válido (sería válido si se reemplaza ra el AND por OR), g. No válido, h. No válido, i. Válido.

17.2 Por medio de un ejemplo mostramos que la junta es conmutativa. La junta A JOIN *B* de las rela ciones A{X,y} y *B[Y,Z]* es una relación con el encabezado *{X,Y,Z} y* un cuerpo que consiste en el conjunto de todas las tupias *{X:x,Y:y,Z:z},* de tal forma que una tupia aparece en A con X valor *x y Y* valor *y,* mientras que una tupia aparece en *B* con *Y* valor *y* y Z valor z. Esta definición es claramente simétrica en A y *B.* Por lo tanto, *A* JOIN *B* = *B* JOIN *A.* |

17.3 Por medio de un ejemplo mostramos que la unión es asociativa. La unión *A* UNION *B* de dos relaciones *A* y *B* es una relación con el mismo encabezado que cada *A* y *B,* y con un cuerpo que con siste en el conjunto de todas las tupias *t* que pertenecen a *A* o a *B,* o a ambas. Por lo tanto, si *C* es otra relación con el mismo encabezado que *A y B:*

- La unión (A UNION *B)* UNION *C* es una relación con el mismo encabezado y con un cuerpo que consiste en todas las tupias *t* que pertenecen a (A UNION B) o a *C,* o a ambas.
- La unión A UNION *(B* UNION C) es una relación con el mismo encabezado y con un cuerpo que consiste en todas las tupias *t* que pertenecen a A o a *(B* UNION *C),* o a ambas.

Estas dos relaciones tienen un mismo encabezado y el cuerpo (en cada caso) es el conjunto de todas las tupias *t,* tales que *t* pertenece al menos a alguna de A, *B* o C. Por lo tanto, las dos relaciones son idénticas.

17.4 Mostramos que la unión se distribuye sobre la intersección:

- Primero, *si te. A* UNION *(B* INTERSECT Q, entonces *t e A o t e B* INTERSECT *C.*
  - *Si te A,* entonces *t* e A UNION *B* y / e A UNION C; por lo tanto *t* e (A UNION *B)* INTER SECT (A UNION O.
  - Si / e *B* INTERSECT *C,* entonces *t* e *B* y *t* e C; por lo tanto *t* e A UNION *ByteA* UNION C; por lo tanto —de nuevo— *t* e (A UNION *B)* INTERSECT (A UNION *Q.*
- En forma alterna, si *t* e (A UNION *B)* INTERSECT (A UNION C), entonces *t* e A UNION *B* y *t* e A UNION C; por lo tanto í e A o *t* e tanto a S como a C; por lo tanto *t* e A UNION *(B* INTERSECT C). |

17.5 Mostramos que A UNION (A INTERSECT *B)* = *A.* Si *t* e A entonces claramente f e A UNION (A INTERSECT *E).* En forma alterna, si *t* e A UNION (A INTERSECT B), entonces uAoíe tanto a A como a *B;* de cualquier forma, í e A.

17.6 Los dos casos condicionales fueron tratados en la sección 17.4. Los casos incondicionales son directos. Mostramos que la proyección no se distribuye sobre la diferencia con el siguiente con traejemplo. Sean A{X,y} y *B{X, Y)* donde cada una contiene sólo una tupia, en particular, las tupias *{X:x,Y:y}* y *{X:x,Y:z},* respectivamente (v * *z).* Entonces (A MINUS *B) {X}* produce una relación que sólo contiene a la tupia *{X:x},* mientras que *A{X]* MINUS *B{X]* produce una relación vacía. |

17.9 Un buen conjunto de dichas reglas puede ser encontrado en la referencia [17.3].

17.10 Un buen conjunto de dichas reglas puede ser encontrado en la referencia [17.3].

17.11

a. Obtener los proveedores "que no son de Londres" que no proporcionan la parte P2.

b. Obtener el conjunto vacío de proveedores.

c. Obtener los proveedores que "no son de Londres", tales que ningún proveedor proporcione menos tipos de partes.

d. Obtener el conjunto vacío de proveedores.

e. No hay simplificación posible.

f. Obtener el conjunto vacío de pares de proveedores.

g. Obtener el conjunto vacío de partes.

h. Obtener los proveedores "que no son de París", tales que ningún proveedor proporcione más tipos de partes.

Observe que determinadas consultas —en particular, las b, d, f y g— pueden ser respondidas directamente a partir de las restricciones de integridad.

17.15 Por razones de procesamiento, el valor verdadero más alto o más bajo es en ocasiones algún tipo de valor *falso;* por ejemplo, el "nombre de empleado" más alto puede ser una cadena con sólo Z y el más bajo una toda en blanco. Las estimaciones de (por ejemplo) el incremento promedio del valor de una columna a la siguiente en secuencia, podría estar falseado si estuviera basado en tales valores falsos.

CAPITULO 18

Información f altante

18.1 INTRODUCCIÓN

En la realidad con frecuencia padecemos por la falta de información; situaciones tales como "fecha de nacimiento desconocida", "conferencista por confirmar", "dirección actual desconocida" son comunes. Por lo tanto, es claro que necesitamos alguna forma de manejar la información faltante dentro de nuestro sistema de base de datos. En la práctica, el enfoque más comúnmente adoptado para este problema —en particular en SQL, y por lo tanto en la mayoría de los productos comerciales— está basado en los **nulos y** en la **lógica de tres valores** (3VL). Por ejemplo, tal vez desconocemos el peso de alguna parte dada, digamos la parte P7, por lo tanto, podríamos decir que el peso de esa parte "es nulo", lo que significa más precisamente que (a) sabemos por supuesto que la parte existe, (b) que tiene un peso y (c) repitiendo, no sabemos cuál es ese peso.

Ampliemos un poco más el ejemplo. Obviamente no podemos poner un valor PESO real en la tupia correspondiente a la parte P7. Por lo tanto, en vez de ello *marcamos* o *indicamos* la posición de PESO en esa tupia como "nulo" y luego interpretamos esa marca o indicador para que signifique, precisamente, que no sabemos cuál es el valor real. Ahora podríamos pensar informalmente que esa posición de PESO "contiene un nulo" o que el valor de PESO "es nulo"; de hecho, en la práctica a menudo hablamos en esos términos. Pero debe quedar claro que tal manera de hablar *es* informal y además no es muy apropiada, ya que decir que el componente PESO de alguna tupia "es nulo", en realidad significa que *la tupia no contiene valor alguno de PESO.* Esta es la razón por la cual se desaprueba la expresión "valor nulo" (la cual es escuchada con mucha frecuencia). El meollo del asunto es precisamente que los nulos no son valores sino marcas o indicadores.

Ahora veremos en la siguiente sección que cualquier comparación escalar en la cual uno de los operandos es nulo, da como resultado un valor de verdad *desconocido,* en lugar de *verdadero o falso.* La justificación para esta situación es la pretendida interpretación de nulo como "valor desconocido": por ejemplo, si el valor de *A* es desconocido, entonces *A* > *B* obviamente también es *desconocido,* **independientemente del valor de *B*** (aun cuando el valor de *B* también sea desconocido). Por lo tanto, observe que no se considera que dos nulos sean iguales entre sí; es decir, si *A* y *B* son nulos, la comparación *A* = *B* da como resultado *desconocido* y no *verdadero.* (Tampoco se considera que no son iguales; es decir, la comparación *A* =£ *B* también da como resultado *desconocido.)* De ahí viene el término "lógica de tres valores". El concepto de nulos, al menos como se entiende usualmente, nos conduce inevitablemente hacia una lógica en la cual hay tres valores de verdad, *verdadero, falso* y *desconocido.*

Antes de continuar, debemos dejar claro que en mi opinión (y también en la de muchos otros autores), los nulos y la 3VL son un gran error y no tienen lugar en un sistema formal y bien

definido como el modelo relational.* Sin embargo, no sería adecuado excluir por completo la explicación de los nulos y la 3VL en un libro de esta naturaleza; a esto se debe el presente capítulo.

La organización del capítulo es la siguiente. Después de esta introducción, en la sección 18.2 evitamos la incredulidad por un momento y describimos (lo mejor que podemos) las ideas básicas con respecto a los nulos y la 3VL, sin criticar demasiado esas ideas. (Es obvio que no es posible criticar sin primero explicar cuáles *son* esas ideas.) Después, en la sección 18.3 explicamos algunas de las consecuencias más importantes derivadas de dichas ideas, para justificar mi posición de que los nulos son un error. La sección 18.4 considera las implicaciones de los nulos para las claves primarias y las externas. La sección 18.5 se desvía un poco para considerar una operación que comúnmente se encuentra en el contexto de los nulos y la 3VL, *es decir* la operación *de. junta externa.* La sección 18.6 considera muy brevemente un enfoque alternativo para la información faltante usando *valores especiales.* La sección 18.7 esboza los aspectos relevantes del SQL en esta materia. Por último, la sección 18.8 presenta un resumen.

Una última indicación preliminar: existen por supuesto muchas razones por las cuales tal vez no podamos asignar un valor de datos genuino en alguna posición dentro de una tupia; el "valor desconocido" es sólo una razón posible. Otras pueden ser "valor no aplicable", "el valor no existe", "valor indefinido", "valor no proporcionado", etcétera [18.5].[f] De hecho, en la referencia [5.2] Codd propone que el modelo relacional debería ser extendido para que incluyera no sólo uno sino dos nulos, uno que indicara "valor desconocido" y el otro "valor no aplicable"; de ahí propone que los DBMS deberían funcionar en términos de lógica no de tres sino de **cuatro** valores. Hemos rechazado en muchos lados esa propuesta [18.5]; en este capítulo limitamos nuestra atención sólo a un tipo de nulo, el llamado nulo de "valor desconocido", al cual nos referiremos a partir de este momento, aunque no invariablemente, como **UNK** (por "unknown").

18.2 UN PANORAMA GENERAL DE LA LÓGICA 3VL

En esta sección describimos brevemente los componentes principales del enfoque 3VL para la información faltante. Comenzamos considerando el efecto de los nulos (específicamente los UNKs) sobre las expresiones lógicas.

---

*Por ejemplo, decir que una determinada tupia de una parte no contiene un valor PESO quiere decir, por definición, que esa tupia en cuestión no es finalmente una tupia de partes. En forma similar, podemos decir que la tupia en cuestión no es un ejemplar del predicado aplicable. La verdad es que el simple acto de tratar de establecer con precisión lo que significa el esquema de los nulos, o lo que debería significar, es suficiente para mostrar por qué la idea no es muy coherente. Una consecuencia es que también resulta difícil expli carlo en forma coherente. Citando la referencia [18.19]: "todo tiene sentido si olvida un poco el asunto y no piensa mucho en él".

*Sin embargo, hacemos notar que no existe "información faltante" como tal en ninguno de estos casos. Por ejemplo, si decimos que la comisión para el empleado Joe "no es aplicable", estamos diciendo, explícita mente, que la propiedad de ganar una comisión no se aplica a Joe y por lo tanto, no hay información fal tante. (Sin embargo, aún existe el caso en que sí, por ejemplo, la "tupia de empleado" de Joe "contiene" un "nulo por no aplicable" en la posición de comisión, entonces esa tupia no es una tupia de empleado; es decir, no es un ejemplar del predicado "empleados".)

Expresiones lógicas

Ya hemos dicho que cualquier comparación escalar en la que cualquiera de los operandos es UNK, da como resultado el valor de verdad *desconocido* en lugar de *verdadero o falso;* y por lo mismo, aquí estamos manejando la lógica de tres valores (3VL). *Desconocido* (al cual a partir de ahora abreviaremos frecuentemente, aunque no invariablemente, como *unk)* es el "tercer valor de verdad". Estas son las tablas de verdad 3VL para AND, OR y NOT (v = *verdadero, f=falso, u = desconocido):*

AND	v	u	f
v	v	u	f
u	u	u	f
f	f	f	f

OR	v	u	f
v	v	v	v
u	v	u	u
f	v	u	f

NOT	
v	f
u	u
f	v

Por ejemplo, supongamos que A = 3, B = 4yCes UNK. Entonces las siguientes expresiones tienen los valores de verdad indicados:

```
A   B AN  B > C     falso
A   B OR  B > C  :  unk
A   B OR  B < C  :  verdadero
N   ( A = C )    :  unk
```

Sin embargo, AND, OR y NOT no son los únicos operadores lógicos que necesitamos [18.11]. Otro operador importante es MAYBE [18.5], el cual tiene una tabla de verdad como la siguiente:

MAYBE	
v	f
u	v
f	f

Para ver por qué MAYBE es necesario, considere la consulta "obtener los empleados que *tal vez,* aunque no se tenga la certeza, sean programadores y que hayan nacido antes del 18 de enero de 1971, con un salario menor a $50,000". Con el operador MAYBE la consulta se puede establecer en forma concisa de la siguiente manera:

```
EMP WHERE MAYBE ( TRABAJO " 'Programador' AND
                  FECHANAC < DATE '1971-1-18' AND
                  SALARIO < 50000.00 )
```

(Hemos supuesto que los atributos TRABAJO, FECHANAC y SALARIO de la varrel EMP son de tipo CHAR, DATE y RATIONAL, respectivamente.) Sin embargo, sin el operador MAYBE la consulta se vería como la siguiente:

```
EMP WHERE ( ISJJNK ( TRABAJO ) AND
            FECHANAC < DATE '1971-1-18' AND
            SALARIO < 50000.00 ) OR   (
    TRABAJO = 'Programador' AND
            ISJJNK ( FECHANAC ) AND
            SALARIO < 50000.00 ) OR   (
    TRABAJO = 'Programador' AND
            FECHANAC < DATE '1971-1-18' AND
            ISJJNK ( SALARIO ) )
    OR    ( ISJJNK ( TRABAJO ) AND
            ISJJNK ( FECHANAC ) AND
            SALARIO < 50000.00 )
```

```
OR   ( ISJJNK ( TRABAJO ) AND
       FECHANAC < DATE '1971-1-18' AND
       ISJJNK ( SALARIO ) ) OR   (
TRABAJO = 'Programador' AND
       ISJJNK ( FECHANAC ) AND
       IS_UNK  (  SALARIO  )  )
OR   ( ISJJNK ( TRABAJO ) AND
       ISJJNK ( FECHANAC ) AND
       ISJJNK ( SALARIO ) )
```

(Hemos dado por hecho la existencia de otro operador de valor de verdad llamado **IS_UNK,** que toma un solo operando de expresión escalar y regresa *verdadero* si ese operando resulta UNK y *falso* en el caso contrario).

Dicho sea de paso, lo anterior no debe ser interpretado como que MAYBE es el *único* operador lógico nuevo necesario para la 3 VL. En la práctica, un operador TRUE_OR_MAYBE, por ejemplo, podría ser muy útil [18.5]. Vea el comentario a la referencia [18.11] en la sección de "Referencias y bibliografía".

EXISTS y FORALL

A pesar de que la mayoría de los ejemplos de este libro están basados en el álgebra en lugar del cálculo, es necesario considerar las implicaciones de la 3VL para los cuantificadores de cálculo EXISTS y FORALL. Como expliqué en el capítulo 7, definimos EXISTS y FORALL como iteraciones de OR y AND, respectivamente. En otras palabras, si (a) *r* es una relación con las tupias *ti, t2,..., tm,* (b) *Ves* una variable de alcance que abarca esa relación y (c) *p(V)* es una expresión lógica en donde *V* aparece como una variable libre; entonces la expresión

```
EXISTS V ( p ( V ) )
```

está definida para ser equivalente a

```
falso OR p ( t1 ) OR ... OR p ( tm )
```

En forma similar, la expresión

```
FORALL V ( p ( V ) )
```

está definida para que sea equivalente a

```
verdadero AND p ( t1 ) AND ... AND p ( tm )
```

Entonces, ¿qué pasa si *p(ti)* da como resultado *unk* para alguna *il* Como ejemplo, hagamos que la relación *r* contenga exactamente las siguientes tupias:

```
( 1.    2,    3 )
( 1,    2,    UNK )(
UNK, UNK, UNK )
```

Por razones de simplicidad, supongamos que: (a) tal como aparecen los tres atributos, de izquierda a derecha, se llaman *A, B* y *C,* respectivamente; (b) cada atributo es de tipo INTEGER. Entonces las siguientes expresiones tienen los valores que se indican:

```
EXISTS V ( V.C > 1  )                  : verdadero
EXISTS V ( V.B > 2  )                  : unk
EXISTS V ( MAYBE (  V.A >  3 ))        : verdadero
EXISTS V ( ISJJNK ( V.C )              ) :  verdadero

FORALL V ( V.A > \  )                  : falso
FORALL V { V.B > 1  )                  : unk
FORALL V ( MAYBE (  V.C >  1 ) )       : falso
```

Expresiones computacionales

Considere la expresión numérica

```
PESO • 454
```

donde PESO representa el peso de alguna parte, digamos P7. ¿Qué pasa si el peso de la parte P7 es UNK? ¿Cuál es entonces el valor de la expresión? La respuesta es que también debe ser considerada como UNK. De hecho, se considera que cualquier expresión numérica dará como resultado UNK cuando cualquiera de los operandos de esa expresión es por sí mismo UNK. Por lo tanto, si sucede por ejemplo que PESO es UNK, entonces todas las expresiones siguientes también resultarán UNK:

```
PESO + 454          454 + PESO          + PESO
PESO - 454          454 - PESO          - PESO
PESO * 454          454 * PESO
PESO / 454          454 / PESO
```

Nota: Tal vez debamos resaltar que el tratamiento anterior de las expresiones numéricas presenta ciertas anomalías. Por ejemplo, la expresión PESO - PESO, que claramente debería producir cero, de hecho produce UNK, y la expresión PESO / 0 que claramente debería dar un error de "división entre cero", también produce UNK (suponiendo en ambos casos, que en primer lugar PESO es UNK). Ignoraremos dichas anomalías mientras no se diga lo contrario.

Otras consideraciones similares se aplican a todos los demás tipos escalares y operadores, con excepción de (a) los operadores de comparación (vea las dos subsecciones anteriores), (b) el operador IS_UNK que expliqué anteriormente y (c) el operador IF_UNK que explicaré en el siguiente párrafo. Por lo tanto, la expresión de cadena de caracteres *A* II *B,* por ejemplo, regresa UNK si *A* es UNK, *B* es UNK o ambos. (Nuevamente, hay determinados casos anómalos cuyos detalles omitimos aquí.)

El operador **IF_UNK** toma dos operandos de expresión escalar y regresa el valor del primer operando, a menos que ese operando dé como resultado UNK, en cuyo caso regresa el valor **del** segundo operando (en otras palabras, el operador en efecto proporciona una forma para convertir un UNK en algún valor no UNK). Por ejemplo, supongamos que se permiten UNKs para el atributo CIUDAD de los proveedores. Entonces, la expresión

```
EXTEND V ADD IFJJNK ( CIUDAD, 'Ciudad desconocida' ) AS CIUDADV
```

produce un resultado en el cual el valor de CIUDADV es "Ciudad desconocida" para cualquier proveedor cuya ciudad esté dada como UNK en V.

Observe, además, que IF_UNK puede ser definido en términos de 1S_UNK. Para ser más específicos, la expresión

```
IFJJNK ( exp1, exp2 )
```

(donde las expresiones *expl* y *exp2* deben ser del mismo tipo) es equivalente a la expresión

```
IF ISJJNK ( expl ) THEN exp2 ELSE expl END IF
```

UNK no es *unk*

Es importante comprender que UNK (el nulo que indica "valor desconocido") y *unk* (el valor de verdad *desconocido)* **no son lo mismo.*** Además, esta situación es una consecuencia inmediata del hecho de que *unk* es un valor (un valor de verdad, para ser precisos) y en cambio, UNK no es un valor en absoluto. Pero seamos más específicos. Supongamos que *X* es una variable de tipo BOOLEAN. Entonces *X* debe tener alguno de los valores *verdadero, falso* o *unk.* Por lo tanto, la instrucción *"X* es *unk"* significa precisamente que **se sabe** que el valor de *X* es *unk.* Por el contrario, la instrucción *"X* es UNK" significa que el valor de *X* **no se conoce.**

¿Puede un dominio contener UNK?

El hecho de que UNK no sea un valor tiene también la consecuencia inmediata de que los UNKs no pueden aparecer en los dominios (los dominios son conjuntos de *valores).* Además, si *fuera* posible que un dominio contuviera un UNK, ¡nunca fallarían las verificaciones de restricción de tipos para ese dominio! Sin embargo, ya que los dominios *no pueden* contener UNKs, una "relación" que incluye un UNK, de hecho **no es una relación,** ni por la definición que dimos en el capítulo 5 ni por la definición original de Codd que aparece en la referencia [5.1]. Posteriormente regresaremos a este punto importante en la sección 18.6.

Expresiones relaciónales

Ahora pondremos nuestra atención en el efecto de los UNKs sobre los operadores del álgebra relational. Por razones de simplicidad nos limitaremos a los operadores de producto, restricción, proyección, unión y diferencia (el efecto de los UNKs sobre los demás operadores puede determinarse a partir de sus efectos en estos cinco).

En primer lugar, el **producto** no es afectado.

En segundo, la operación de **restricción** es redefinida (ligeramente) para que regrese una relación cuyo cuerpo contenga solamente aquellas tupias para las que la condición de restricción dé como resultado *verdadero;* es decir, ni *falso* ni *unk. Nota:* Anteriormente dimos por hecho esta redefinición en nuestro ejemplo para MAYBE de la subsección "Expresiones lógicas".

Después, la **proyección.** Por supuesto, la proyección involucra la eliminación de tupias redundantes duplicadas. Ahora, en la lógica de *dos* valores (2VL) convencional, dos tupias están duplicadas si, y sólo si, tienen los mismos atributos y los atributos correspondientes tienen valores iguales. Sin embargo, en la 3VL algunos de esos valores de atributo podrían ser UNK, y

---

: Sin embargo, en SQL3 se considera que sí lo son (vea el Apéndice B).

UNK (como hemos visto) no es igual a *cualquier cosa,* ni siquiera a sí mismo. Entonces, ¿estamos obligados a concluir que una tupia que contiene un UNK nunca puede ser un duplicado de nada?, ¿ni siquiera de ella misma?

De acuerdo con Codd, la respuesta a esta pregunta es *no:* dos UNKs, aunque no sean iguales entre sí, se consideran "duplicados" uno del otro para efectos de eliminación de tupias duplicadas [13.6].* La contradicción aparente se explica de la siguiente manera:

> [La prueba de igualdad] para eliminar duplicados se encuentra... a un nivel de detalle menor que la prueba de igualdad en la evaluación de condiciones de recuperación. Por lo tanto, es posible adoptar una regla diferente.

Dejamos a su juicio determinar si estos argumentos son razonables o no. De cualquier forma, aceptémoslos por ahora y también aceptemos la siguiente definición:

- Dos tupias son **duplicados** si, y sólo si, (a) tienen los mismos atributos y (b) para cada atributo las dos tupias tienen ya sea el mismo valor o ambas tienen un UNK en esa posición del atributo.

Con esta definición extendida de las "tupias duplicadas", la definición original de proyección se aplica sin modificaciones.

En forma similar, la **unión** involucra la eliminación de tupias redundantes duplicadas y se aplica la misma definición de tupias duplicadas. Por lo tanto, definimos la unión de dos relaciones *rl* y *r2* (del mismo tipo) como la relación *r* (nuevamente del mismo tipo) cuyo cuerpo consiste en todas las tupias *t* posibles tales que *t* sea un duplicado de alguna tupia de *rl* o de *rl* (o de ambas).

Por último, aunque no involucra eliminación alguna de duplicados como tales, la **diferencia** se define en forma similar; es decir, una tupia *t* aparece en *rl* MINUS *r2* si, y sólo si, es un duplicado de alguna tupia de *rl* y no es un duplicado de ninguna tupia de *r2.* (Por supuesto, por lo que se refiere a la **intersección,** ésta no es primitiva, pero para completar hacemos notar que también se define en forma similar; es decir, una tupia *t* aparece en *rl* INTERSECT *rl* si, y sólo si, es un duplicado de alguna tupia de *rl* y alguna tupia de *r2).*

Operaciones de actualización

Hay dos puntos generales a tratar bajo este encabezado:

1. Si el atributo A de la varrel *R* permite UNKs, y si una tupia es insertada en *R* sin que se proporcione un valor para A, el sistema colocará automáticamente un UNK en la posición de A de esa tupia. Si el atributo A de la varrel *R* no permite UNKs, resulta un error intentar crear una tupia en *R* mediante INSERT o UPDATE en donde la posición de A es UNK. (Por supuesto, en ambos casos estamos suponiendo que no se ha definido un valor predeterminado noUNKparaA.)

---

* La referencia [13.6] fue el primer artículo de Codd para explicar el problema de la información faltante (aunque ese problema no fue el tema principal del artículo; vea el capítulo 13). El artículo propone, entre otras cosas, versiones "tal vez" de los operadores junta-®, selección-® (es decir, restricción) y división (vea el ejercicio 18.4), así como versiones "externas" de los operadores de unión, intersección, diferencia, junta-© y junta natural (vea la sección 18.5).

2. Como de costumbre, es un error intentar crear una tupia duplicada mediante INSERT o UPDATE. Aquí, la definición de "tupias duplicadas" es igual a la de la subsección anterior.

Restricciones de integridad

Como expliqué en el capítulo 8, una restricción de integridad es básicamente una expresión lógica que no debe dar como resultado *falso*. Por lo tanto, observe que la restricción no se viola si ésta da como resultado *unk* (de hecho, lo hemos estado dando por hecho anteriormente en esta sección, al considerar las restricciones de tipo). Técnicamente debemos decir que en este caso *no se sabe* si la restricción es violada; pero así como consideramos a *unk* como *falso* en el caso de una cláusula WHERE, también lo consideramos *verdadero* en el caso de una restricción de integridad (hablando un poco a la ligera).

18.3 ALGUNAS CONSECUENCIAS DEL ESQUEMA ANTERIOR

El enfoque 3VL, tal como lo describí en la sección anterior, tiene varias consecuencias lógicas, aunque no todas son obvias inmediatamente. En esta sección explicaremos algunas de estas consecuencias y su significado.

Transformación de expresiones

Primero, observamos que varias expresiones que siempre dan como resultado *verdadero* en 2VL, no necesariamente siempre dan como resultado *verdadero* en 3VL. Estos son algunos ejemplos. Observe que la lista no es muy extensa.

- *La comparación x = xno necesariamente da* verdadero

  En 2VL cualquier valor *x* siempre es igual a sí mismo. Sin embargo, en 3VL *x* no es igual a sí mismo si resulta ser UNK.

- *La expresión lógica* p *OR NOT (p) no necesariamente da* verdadero

  En este caso, *p* es una expresión lógica. Ahora, en 2VL la expresión *p* OR NOT(p) da como resultado *verdadero*, independientemente del valor *dep*. Sin embargo, si en *3VLp* llega a dar como resultado *unk*, la expresión general da como resultado *unk* OR NOT(Mn&); es decir, *unk* OR *unk*, lo cual se reduce a *unk* y no a *verdadero*.

  Este ejemplo en particular nos conduce a una propiedad contraintuitiva bien conocida de 3VL, la cual ilustramos de la siguiente manera: si formulamos la consulta "obtener todos los proveedores en Londres" seguida de la consulta "obtener todos los proveedores que no están en Londres" y tomamos la unión de los dos resultados, *no* necesariamente obtenemos todos los proveedores. En vez de ello, necesitamos incluir "todos los proveedores que *pudieran estaren* Londres".*

---

* Como lo sugiere esta explicación, una expresión que siempre dé como resultado verdadero en 3VL; —es decir, la analogía en 3VL de la expresión 2VL *p* OR NOT(p)- es *p* OR NOT(p) OR MAYBE(p).

Por supuesto, el punto en este ejemplo es, que mientras los dos casos, "la ciudad es Londres" y "la ciudad no es Londres", son mutuamente exclusivos y agotan por completo el rango de posibilidades en la realidad, la base de datos *no* contiene la realidad, sino que en vez de ello contiene sólo **un conocimiento acerca** de la realidad. Y existen no dos, sino tres casos posibles que se refieren al conocimiento de la realidad; en el ejemplo son: "se sabe que la ciudad es Londres", "se sabe que la ciudad no es Londres" y "no se conoce la ciudad". Además, como señala la referencia [18.6], obviamente no podemos preguntarle al sistema acerca de la realidad, sino que sólo podemos preguntarle acerca de su *conocimiento* de la realidad tal como está representado en la base de datos. La naturaleza contraintuitiva del ejemplo se deriva de una confusión sobre los dominios: el usuario está pensando en términos del dominio que es la realidad, pero el sistema está operando en términos del dominio que es *su conocimiento acerca de* esa realidad.

Nota: Sin embargo, a mí me parece que esa confusión sobre los dominios es una trampa en la que podemos caer muy fácilmente. Observe que *cada una de las consultas* mencionadas en los capítulos anteriores de este libro (ejemplos, ejercicios, etcétera) ha sido formulada en términos de "la realidad" y no en términos del "conocimiento acerca de la realidad"; y este libro no es el único en tener esto en consideración.

- *La expresión* r *JOIN* r *no necesariamente da* r

  En 2VL, la formación de la junta de la relación ***r*** con sí misma siempre da como resultado la relación original *r* (es decir, la junta es *idempotente).* Sin embargo, en 3VL una tupia con un UNK en cualquier posición no se juntará consigo misma, ya que la junta, a diferencia de la unión, está basada en la prueba de igualdad del "tipo recuperación" y no en la del "tipo duplicado".

- *INTERSECT ya no es un caso especial de JOIN*

  Esto es consecuencia de que (de nuevo) la junta está basada en la prueba de igualdad por tipo-recuperación, mientras que la intersección está basada en la prueba de igualdad por tipo-duplicado.

- A = B y B = C *juntos no implican* A = C

  Más adelante, en la subsección "El ejemplo de los departamentos y empleados", ilustramos ampliamente este punto en particular.

En resumen, muchas de las equivalencias que son válidas en 2VL se rompen en 3VL. Una consecuencia muy seria de tales rupturas es la siguiente. Como expliqué en el capítulo 17, generalmente las equivalencias simples como ***r*** JOIN *r* = *r* están basadas en las diversas **leyes** de **transformación** usadas para convertir consultas en formas más eficientes. Además, esas leyes no sólo son usadas por el *sistema* (cuando hace una optimización), sino también por los *usuarios* (cuando intentan definir la "mejor" forma de plantear una consulta dada). Y si las equivalencias no son válidas, entonces las leyes de transformación no lo son. Y si las leyes no son válidas, las transformaciones tampoco lo son. Y si las transformaciones no son válidas, obtendremos **respuestas erróneas** del sistema.

El ejemplo de los departamentos y empleados

Para ilustrar el problema de las transformaciones incorrectas, explicamos detalladamente un ejemplo específico. (El ejemplo fue tomado de la referencia [18.9], y por razones que aquí no

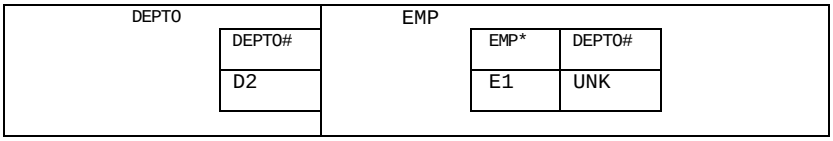


Figura 18.1 La base de datos de departamentos y de empleados.

son importantes, está basado en cálculo relacional en vez del álgebra relacional.) Supongamos que tenemos la base de datos de departamentos y empleados que muestra la figura 18.1. Considere la expresión

```
DEPTO.DEPT0# = EMP.DEPTO* AND EMP.DEPTO# • DEPTO# ( ' D1 " )
```

(la cual, por supuesto podría ser parte de una consulta); en ella DEPTO y EMP son variables de alcance implícitas. Para las únicas tupias que existen en la base de datos, esta expresión da como resultado *unk* AND *unk;* es decir, *unk.* Sin embargo, un "buen" optimizador observará que la expresión es de la forma *a* = *b* AND *b* = *cy* por lo tanto, inferirá que *a* = c, lo que le llevará a añadir un término de restricción adicional *a* = *c* a la expresión original (como mencioné en el capítulo 17, sección 17.4), produciendo

```
DEPTO. DEPTO* = EMP.DEPTO# AND EMP.DEPTO* = DEPT0# ( ' D1 ' )
                           AND DEPTO.DEPT0# = DEPTO# ( 'D1' )
```

Esta expresión modificada ahora da como resultado *unk* AND *unk* AND *falso;* es decir, *falso* (para las dos únicas tupias en la base de datos). Por lo tanto, se desprende que la consulta

```
  EMP.EMP* WHERE EXISTS DEPTO ( NOT ( DEPTO. DEPT0# ■ EMP. DEPTO*
AND EMP.DEPTO# = DEPTO* ( ' D1 ' ) ) )
```

regresará al empleado El si es "optimizada " en el sentido anterior y no lo hará en caso contrario. En otras palabras, la "optimización" no es válida. Por lo tanto, vemos que determinadas optimizaciones que son perfectamente válidas y útiles bajo la 2VL convencional ya no lo son bajo 3VL.

Note las implicaciones de lo anterior para extender un sistema 2VL con el fin de que soporte 3VL. En el mejor de los casos, es probable que tal extensión requiera de algunos arreglos al sistema existente (ya que quizás existan algunas porciones del código optimizador que no son válidas); y en el peor de los casos, introducirá errores. En términos más generales, observe las implicaciones de extender un sistema que soporta una lógica de *n* valores a otro que soporta (n + 1) valores, para cualquier número de *n* mayor que 1; se presentarán otras dificultades similares para cada valor discreto de *n.*

El tema de la interpretación

Ahora, analicemos más cuidadosamente el ejemplo de departamentos y de empleados. Puesto que en la realidad el empleado El corresponde a algún departamento, el UNK significa un valor real (digamos *d).* Ahora *d* es o no es DI. Si lo es, la expresión original

```
DEPTO.DEPTO# = EMP.DEPTO# AND EMP.DEPTO# = DEPTO# ( ' D1 ' )
```

da como resultado (para los datos proporcionados) *falso,* ya que el primer término resulta *falso.* En forma alterna, si *d* no es D1, entonces la expresión también da como resultado (para los datos proporcionados) *falso,* ya que el segundo término resulta *falso.* En otras palabras, la expresión original siempre *es falsa* en la realidad, **independientemente del valor que representa UNK.** Por lo tanto, ¡el resultado que es correcto de acuerdo a la lógica de tres valores y el resultado que es correcto en la realidad no es el mismo! Es decir, la lógica de tres valores no se comporta de acuerdo con la realidad; o lo que es lo mismo, la 3VL parece no tener una interpretación sensible a la manera en que funciona la realidad.

Nota: Este asunto de la interpretación está muy lejos de ser el único problema que se presenta con los nulos y la 3VL (vea las referencias [18.1] a [18.11], para una amplia explicación de otros problemas). Ni siquiera es el más importante (vea la siguiente subsección). Sin embargo, tal vez es uno de los que tiene mayor importancia práctica, ya que, en mi opinión esto es extraordinario.

De nuevo los predicados

Supongamos que la relación que representa el valor actual de la varrel EMP contiene sólo dos tupias (E2, D2) y (El, UNK). La primera corresponde a la proposición "hay un empleado identificado como E2 en el departamento identificado como D2". La segunda corresponde a la proposición "hay un empleado identificado como El". (Recuerde que decir que una tupia "contiene un UNK", en realidad significa que la tupia de hecho contiene nada en absoluto en la posición aplicable; por lo tanto, la "tupia" (El, UNK) —si fuera realmente una tupia, lo que de por síes una noción dudosa— debería en realidad ser considerada simplemente de la forma (El).) En otras palabras, las dos tupias son ejemplares de *dos predicados diferentes* y la "relación" no es ninguna relación sino que es (vagamente) un tipo de unión de dos relaciones diferentes con dos encabezados particularmente diferentes.

Ahora, tal vez podríamos sugerir que la relación puede ser rescatada afirmando que en realidad sólo existe un predicado, aquél que involucra un OR:

> Hay un empleado identificado como E# en el departamento identificado como D# OR hay un empleado identificado como E#.

Sin embargo, observe que ahora (gracias a la Suposición de Mundo Cerrado) ¡la relación tendrá que contener una "tupia" de la forma (Eí, UNK) para todos los empleados E¡! Es terrible contemplar la generalización de este intento de rescatar una "relación" que "contiene UNK" en varios "atributos" diferentes. (Y en cualquier caso, la "relación" que resulte no será una relación; vea el siguiente párrafo.)

Para expresarlo de otra forma: si el valor de un atributo dado dentro de una tupia dada dentro de una relación dada "es UNK", entonces (repitiendo) esa posición de atributo de hecho contiene **nada en absoluto...** lo cual implica que el "atributo" no es un atributo, la "tupia" no es una tupia, la "relación" no es una relación y los fundamentos de lo que estamos haciendo (independientemente de lo que pueda ser) ya no son teoría de relaciones matemáticas. En otras palabras, los UNK y la 3VL *minan todos los fundamentos del modelo relacional*.

3.4 LOS NULOS Y LAS CLAVES

Nota: Ahora descartamos el término UNK (en su mayor parte) y regresamos por razones históricas a la terminología más tradicional de "nulos".

A pesar del mensaje de la sección anterior, el hecho es que al momento de la publicación de este libro los nulos y la 3VL son soportados por la mayoría de los productos. Además, tal soporte tiene implicaciones importantes, en particular para las claves. Por lo tanto, en esta sección analizamos brevemente esas implicaciones.

Claves primarias

Como expliqué en la sección 8.8, el modelo relacional ha requerido históricamente que (al menos en el caso de las varrels base) se elija una sola clave candidata como clave *primaria* para la varrel en cuestión. Decimos entonces que las claves candidatas restantes, en caso de haberlas, son claves *alternas.* Y luego, junto con el concepto de clave primaria, el modelo ha incluido históricamente la siguiente "metarrestricción" o regla (la regla de *integridad de la entidad):*

- **Integridad de la entidad:** no está permitido que ningún componente de la clave primaria de cualquier varrel base acepte nulos.

Las razones para esta regla son más o menos las siguientes: (a) las tupias en las relaciones base representan entidades en la realidad; (b) las entidades en la realidad son identificables por definición; (c) por lo tanto, sus contrapartes en la base de datos también deben ser identificables; (d) los valores de la clave primaria sirven como esos identificadores en la base de datos; (e) por lo tanto, los valores de clave primaria no pueden estar "faltantes". Se presentan estos puntos:

1. En primer lugar, a menudo se piensa que la regla de integridad de la entidad dice algo así como "los valores de clave primaria deben ser únicos", pero no es así. (Por supuesto, es cierto que los valores de la clave primaria deben ser únicos, pero ese requerimiento está implí cito en la definición del concepto de clave primaria *por sí mismo.)*
2. Luego, observe que la regla se aplica sólo a las claves *primarias;* aparentemente, las claves *alternas* pueden permitir nulos. Pero si *AK* es una clave alterna que permite nulos, entonces no se habría podido escoger a *AK* como clave primaria, debido a la regla de integridad de la entidad; entonces ¿en qué sentido *AK* era una clave "candidata"? En forma alterna, si tene mos que decir que las claves alternas no pueden permitir nulos, entonces la regla de integri dad de la entidad se aplica a *todas las claves candidatas* y no sólo a la clave primaria. De cualquier forma, parece que hay algo erróneo en la regla tal como está expresada.
3. Por último, observe que la regla de integridad de la entidad se aplica solamente a las *va rrels base;* aparentemente las demás varrels pueden tener una clave primaria en la que se permitan los nulos. Como ejemplo trivial y obvio, considere la proyección de una varrel *R* sobre cualquier atributo *A* que permita nulos. Entonces esta regla viola el *Principio de intercambiabilidad* (de las varrels base y las derivadas). En mi opinión, éste sería un fuerte argumento para rechazarla (aunque no involucrara nulos, un concepto que de todas formas rechazamos).

Nota: Supongamos que estamos de acuerdo en eliminar la idea de los nulos por completo y en vez de ello, usamos *valores especiales* para representar la información faltante (de hecho, es como lo hacemos en la realidad; vea la sección 18.6). Entonces, tal vez queramos conservar una versión modificada de la regla de integridad de la entidad —"A ningún componente de la clave primaria de cualquier varrel base se le permite aceptar dichos valores especiales"— como *lineamiento*, pero *no* como ley inviolable (en forma similar a como las ideas de la normalización sirven como lineamientos pero no como leyes inviolables). La figura 18.2 muestra un ejemplo (tomado de la referencia [5.7]) de una varrel base llamada ESTUDIO para la cual, tal vez, queramos violar ese lineamiento; este ejemplo representa los resultados de un estudio de salarios que muestra el salario promedio, el máximo y el mínimo por año de nacimiento para una determinada muestra de población (AÑO_NACIMIENTO es la clave primaria). Y la tupia con el valor especial **"????"** en AÑO_NACIMIENTO representa a las personas que se negaron a responder a la pregunta "¿Cuando nació usted?".

ESTUDIO

AÑO_NACIMIENTO	SALPROM	SALMAX	SALMIN
1960	85K	130K	33K
1961	82K	125K	32K
1962	77K	99K	32K
1963	78K	97K	35K
1970	29 K	35K	12K
????	56K	117K	20K

Figura 18.2 La varrel base ESTUDIO (valores de ejemplo).

Claves externas

Nuevamente considere la base de datos de departamentos y empleados de la figura 18.1. Tal vez no lo haya notado anteriormente, pero en forma deliberada no mencionamos que el atributo DEPTO# de la varrel EMP en esa figura era una clave externa. Pero ahora supongamos que lo es. Entonces queda claro que la regla de integridad referencial necesita cierto refinamiento, ya que ahora las claves externas deben, en apariencia, permitir nulos y obviamente los valores nulos de la clave externa violan la regla tal como la establecimos originalmente en el capítulo 8.

- **Integridad referencial** *(versión original):* la base de datos no debe contener ningún valor de clave externa que no concuerde.

De hecho, podemos mantener la regla como está establecida, siempre y cuando extendamos adecuadamente la definición del término "valor de clave externa que no concuerde". Para ser específicos, definimos ahora un "valor de clave externa que no concuerde" como un "valor de clave externa **no nulo",** en alguna varrel referente, para la cual no exista un valor que concuerde en la clave candidata relevante en la varrel relevante a la que se hace referencia.

Se presentan estos puntos:

1. El que se permita o no que cualquier clave externa acepte nulos tendrá que ser especificado como parte de la definición de la base de datos. (Por supuesto, lo mismo es cierto para los atributos en general, independientemente de si forman parte de alguna clave externa o no).

2. La posibilidad de que las claves externas puedan aceptar nulos hace que se presente la posi bilidad de otra acción referencial, SET NULL, la cual puede ser especificada en una regla DELETE o UPDATE de clave externa. Por ejemplo:

```
VAR VP BASE RELATION { ... } ...
    FOREIGN KEY { V# } REFERENCES V
                      ON DELETE SET NULL ON
                      UPDATE SET NULL ;
```

   Con estas especificaciones, una operación DELETE sobre la varrel de proveedores colocará la clave externa en nulo para todos los envíos que concuerden y luego borrará todos los proveedores aplicables; de manera similar, una operación UPDATE sobre el atributo V# en la varrel de proveedores pondrá la clave externa en nulo para todos los envíos que concuerden y luego actualizará los proveedores aplicables. *Nota:* SET NULL puede especificarse solamente para una clave externa que, por supuesto, en primer lugar acepte nulos.

3. Por último, observamos que la aparente "necesidad" de permitir nulos en las claves exter nas puede evitarse con un diseño adecuado de la base de datos [18.20]. Considere nueva mente, por ejemplo, los departamentos y empleados. Si realmente es posible que no se conozca el número de departamento para determinados empleados, entonces (como sugerí casi al final de la sección anterior) seguramente sena mejor no incluir a DEPTO# en la va rrel EMP, sino tener aparte una varrel ED (por ejemplo) con los atributos EMP# y DEPTO# para representar el hecho de que el empleado especificado está en el departamento especi ficado. Luego, el hecho de que un empleado determinado tenga un departamento descono cido puede representarse por la *omisión* de una tupia para ese empleado en la varrel ED.

I LA JUNTA EXTERNA (UNA OBSERVACIÓN)

En esta sección nos desviaremos brevemente para tratar una operación conocida como **junta externa** (vea las referencias [18.3], [18.4], [18.7] y [18.14] a [18.16]). La junta externa es una forma extendida de la operación de junta normal o *interna.* Difiere con respecto a la junta interna en que las tupias de una relación que no tienen contraparte en la otra, aparecen en el resultado con nulos en las posiciones de atributo de la otra, en lugar de sólo ignorarlas como sucede normalmente. No es una operación primitiva ya que, por ejemplo, la siguiente expresión podría ser usada para construir la junta externa de proveedores y envíos sobre números de proveedor (suponiendo para efectos del ejemplo que "NULL" es una expresión escalar válida):

```
( V JOIN VP )
UNION
( EXTEND ( ( V  { V# } MINUS VP { V# } ) JOIN V )
                                ADD NULL AS P#, NULL AS CANT )
```

El resultado incluye tupias para proveedores que no proporcionan partes, extendidas con nulos en las posiciones P# y CANT.

Examinemos este ejemplo más de cerca. Consulte la figura 18.3. En esta figura la parte superior muestra algunos valores de datos de ejemplo, la parte media muestra la junta interna correspondiente y la parte inferior muestra la junta externa correspondiente. Como lo indica la figura, la junta interna "pierde información" (hablando *muy* a la ligera) para los proveedores que

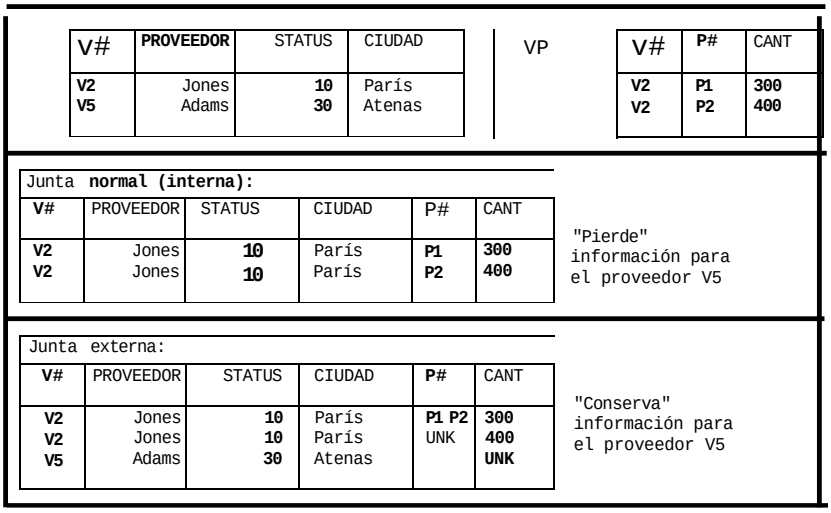


Figura 18.3 Junta interna vs. junta externa (ejemplo).

no proporcionan partes (el proveedor V5 en el ejemplo) y en cambio, la junta externa "conserva" tal información. Esta distinción es todo el punto de la junta externa.

Ahora el problema que trata de resolver la junta externa (es decir, el hecho de que la junta interna a veces "pierde información") es ciertamente un problema importante. Por lo tanto, algunos autores podrían argumentar que el sistema debería proporcionar soporte directo y explícito para la junta extema, en lugar de que el usuario tenga que ingeniárselas para lograr el efecto deseado. En particular, Codd considera ahora que la junta extema es una parte intrínseca del modelo relacional [5.2]. Sin embargo, nosotros no apoyamos esta posición por las siguientes razones, entre otras:

- En primer lugar, por supuesto, la operación involucra nulos y nosotros nos oponemos com pletamente a los nulos por varias buenas razones.
- Segundo, observe que existen diferentes variedades de junta externa: *junta* © izquierda, derecha y completa, así como junta *natural* externa izquierda, derecha y completa. (Las jun tas "izquierdas" conservan información del primer operando, las "derechas" del segundo y las "completas" de ambos; el ejemplo de la figura 18.3 es una junta izquierda —una junta natu ral extema izquierda, para ser más precisos—.) Observe, además, que no hay una forma muy directa para derivar las juntas naturales externas a partir de las juntas © extemas [18.7]. Como resultado, no queda claro exactamente qué juntas extemas necesitan soporte explícito.
- Luego, la cuestión de la junta extema está lejos de ser tan trivial como podría sugerirlo el ejemplo de la figura 18.3. De hecho, como lo marca la referencia [18.7], la junta externa está afectada por varias *propiedades desagradables,* las cuales implican que agregar soporte de junta externa a los lenguajes existentes (en particular al SQL) es algo difícil de hacer

armoniosamente. Varios productos DBMS han intentado resolver este problema y han fallado tristemente (es decir, han tropezado con esas propiedades desagradables). Vea la referencia [18.7] para una explicación más amplia de este tema.

- Por último, los *atributos con valores de relación* proporcionan, de cualquier forma, un enfoque alternativo al problema; un enfoque que no involucra nulos ni una junta externa y es de hecho (en mi opinión) una solución más elegante. Por ejemplo, tomando los valores de la parte superior de la figura 18.3, la expresión:

```
WITH ( V RENAME V# AS X ) AS Y : EXTEND
Y ADD ( VP WHERE V# = X ) AS PC
```

produce los resultados que muestra la figura 18.4.

V#	PROVEEDOR	STATUS	CIUDAD	PC
V2	Jones	10	París	P# | CANT: P1 | 300; P2 | 400
V5	Adams	30	Atenas	P# | CANT: (vacío)

Figura 18.4 Conservación de la información para el proveedor V5 (una mejor forma).

Observe en particular que en la figura 18.4 el conjunto vacío de partes proporcionadas por el proveedor V5 está representado por un conjunto vacío, y no (como en la figura 18.3) por algún "nulo" extraño. Representar un conjunto vacío mediante un conjunto vacío parece, obviamente, una buena idea. De hecho, *no habría necesidad de ninguna junta externa* si los atributos con valores de relación fueran soportados adecuadamente.

Para continuar un poco más con este asunto: ¿cómo se supone que debemos *interpretar* los nulos que aparecen en el resultado de una junta externa? ¿Qué significan en el ejemplo de la figura 18.3? Ciertamente no significan "valor desconocido" o "valor no aplicado", sino que, de hecho, la única interpretación que tiene algún sentido lógico es precisamente "el valor es el conjunto vacío". Para una mayor explicación de este punto, vea la referencia [18.7].

Cerramos esta sección comentando que también es posible definir versiones "externas" de otras operaciones del álgebra relacional, específicamente las operaciones de unión, intersección y diferencia [13.6]; de nuevo Codd considera ahora al menos una de éstas, llamada *unión externa*, como parte del modelo relacional [5.2]. Dichas operaciones permiten realizar uniones (por citar un ejemplo) entre dos relaciones aun cuando no sean del mismo tipo; básicamente funcio-

nan extendiendo cada operando para que incluya aquellos atributos que son característicos del otro (de tal forma que los operandos son ahora del mismo tipo), poniendo nulos en cada tupia para todos esos atributos añadidos y luego realizando una unión, intersección o diferencia normal según sea el caso.* Sin embargo, no explicamos detalladamente estas operaciones por las siguientes razones:

- Está garantizado que la intersección externa regresa una relación vacía, con excepción del caso especial en el cual las relaciones originales son, en primer lugar, del mismo tipo, en cuyo caso degenera hacia la intersección normal.
- Está garantizado que la diferencia externa regresa su primer operando, con excepción del caso especial en el cual las relaciones originales son, en primer lugar, del mismo tipo, en cuyo caso degenera hacia la diferencia normal.
- Las uniones externas tienen problemas *importantes* de interpretación (son peores que los que se presentan con la junta externa). Vea la referencia [18.2] para una explicación adicional.

18.6 VALORES ESPECIALES

Hemos visto que los nulos echan a perder el modelo relacional. De hecho, vale la pena señalar que ¡el modelo relacional funcionó perfectamente bien durante diez años sin nulos!, ya que el modelo fue definido por primera vez en 1969 [5.1] y los nulos se añadieron hasta 1979 [13.1].

Por lo tanto, como sugerí en la sección 18.4, supongamos que estamos de acuerdo en desechar la idea completa de los nulos y en su lugar usamos *valores especiales* para representar la información faltante. Observe que el uso de valores especiales es exactamente lo que hacemos en la realidad. Por ejemplo, podemos usar el valor especial "?" para indicar las horas trabajadas por determinado empleado cuando el valor real es desconocido por alguna razón.[1] Por lo tanto, la idea general es simplemente usar un valor especial adecuado, distinto a todos los valores normales para el atributo en cuestión, cuando no se puede usar un valor normal. Observe que el valor especial debe ser un valor del dominio aplicable; por lo tanto, en el ejemplo de "horas trabajadas", el tipo del atributo HORAS_TRABAJADAS no es solamente de enteros, sino de enteros más lo que sea el valor especial.

Ahora, quisiéramos ser los primeros en admitir que el esquema anterior no es muy elegante, pero tiene la gran ventaja de que *no mina los fundamentos lógicos del modelo relacional.* Por lo tanto, en el resto de este libro simplemente ignoraremos la posibilidad del soporte de nulos (con excepción de determinados contextos específicos de SQL, en donde es inevitable hacer referencias ocasionales a los nulos). Vea la referencia [18.12] para una descripción detallada del esquema de valores especiales.

---

* Esta explicación se refiere a las operaciones tal como se definieron originalmente [13.6]. La referencia [5.2] cambió un poco las definiciones; usted será remitido a ese libro para los puntos específicos (en caso de que le interese)

[+] Observe que una cosa que nosotros *no* hacemos es usar un nulo para este propósito. En !a realidad no existe cosa alguna que sea un nulo en el mundo real.

18.7 PROPIEDADES DE SQL

El soporte de SQL para los nulos y la 3 VL sigue ampliamente los lineamientos del enfoque descrito anteriormente en las secciones 18.1 a 18.5. Por lo tanto cuando, por ejemplo, SQL aplica una cláusula WHERE a alguna tabla *T,* ésta elimina todas las filas de *T*para las cuales la expresión que está en esa cláusula WHERE dé como resultado *falso* o *unk* (es decir, no *verdadero).* En forma similar, cuando aplica una cláusula HAVING a alguna "tabla agrupada" *G* (vea el apéndice A), ésta elimina todos los grupos de *G* para los cuales la expresión de esa cláusula HAVING dé como resultado *falso* o *unk* (es decir, no *verdadero).* Por lo tanto, en adelante, simplemente pondremos atención a determinadas características de *3VL* que son específicas de SQL *en sí mismo,* en lugar de ser una parte intrínseca del enfoque *3VL* como mencioné anteriormente.

Nota: Las implicaciones y ramificaciones del soporte de SQL para los nulos, son muy complejas. Para obtener información adicional consulte el documento del estándar oficial [4.22] o el tratamiento del tutorial detallado que se da en la referencia [4.19].

Definición de datos

Como expliqué en la sección 5.5 del capítulo 5, por lo general las columnas de las tablas base tienen asociado un valor *predeterminado* y este valor a menudo está definido explícita o implícitamente para que sea nulo. Además, las columnas de las tablas base siempre *permiten* nulos, a menos que haya una restricción de integridad —probablemente NOT NULL— para la columna en cuestión que los prohiba explícitamente. La representación de los nulos depende de la implementación, pero debe ser de forma tal que el sistema pueda distinguir a los nulos de todos los demás valores posibles que no lo sean (¡aunque la comparación "nulo =£ *x"* no dé *verdadero* !).

Expresiones de tabla

Recuerde que en la sección 7.7 del capítulo 7 explicamos que a SQL le fue añadido soporte explícito para JOIN con el estándar de SQL/92. Si a la palabra reservada JOIN se le pone el prefijo LEFT, RIGHT o FULL (seguido opcionalmente por la palabra OUTER en cada caso), entonces la junta en cuestión es una junta *externa.* Estos son algunos ejemplos:

```
V LEFT JOIN VP ON V.V# = VP.V#
V LEFT JOIN VP USING ( V# )
V LEFT NATURAL JOIN VP
```

Estas tres expresiones son en efecto equivalentes, salvo que la primera produce una tabla con dos columnas idénticas (llamadas ambas V#) y la segunda y la tercera producen una tabla con una sola de estas columnas.

SQL también soporta una aproximación a la unión externa, a la que llama *junta de unión.* Los detalles están fuera del alcance de este libro.

Expresiones condicionales

Como observamos en el capítulo 8, las expresiones condicionales son la analogía de SQL para lo que hemos estado llamando expresiones *lógicas* (en el apéndice A las explico en detalle). No es sorprendente que dichas expresiones sean la parte de SQL más drásticamente afectada por los nulos y la 3VL. En este punto sólo hacemos algunos comentarios pertinentes.

- *Pruebas para nulo:* SQL proporciona dos operadores especiales de comparación, IS NULL e IS NOT NULL, para determinar la presencia o ausencia de nulos. La sintaxis es:

  ```
  <constructor de fila>  IS  [NOT]   NULL
  ```

  (vea el apéndice A para los detalles de *<constructor defila>).* Hay una trampa para quien no esté prevenido: las dos expresiones r IS NOT NULL y NOT(r IS NULL) *no* son equivalentes. Para una mejor explicación vea la referencia [4.19].

- *Pruebas para* verdadero, falso, desconocido: *úp* es una expresión condicional entre parén tesis, entonces las siguientes también son expresiones condicionales:

  ```
  p IS [ NOT ] TRUE p
  IS [ NOT ] FALSE p
  IS [ NOT ] UNKNOWN
  ```

  El significado de estas expresiones es el indicado en la siguiente tabla de verdad:

p	*verdadero*	*falso*	*unk*
p I TRUE	*verdadero*	*falso*	*falso*
p Î NOT TRUE	*falso*	*verdadero*	*verdadero*
p I FALSE	*falso*	*verdadero*	*falso*
p I NOT FALSE	*verdadero*	*falso*	*verdadero*
p I UNKNOWN	*falso*	*falso*	*verdadero*
p I NOT UNKNOWN	*verdadero*	*verdadero*	*falso*

  Por lo tanto, observe que las expresiones *p* IS NOT TRUE y NOT *p* no son equivalentes. *Nota:* La expresión *p* IS UNKNOWN corresponde a nuestro MAYBE(p).

  Condiciones MATCH: La sintaxis de *<condición MATCH>* (vea el apéndice A) incluye una opción PARTIAL y otra FULL (que no se muestran ni se tratan en el apéndice A) que pueden afectar el resultado, si hay nulos presentes:

  ```
  <constructor de fila> MATCH [ UNIQUE ]
                        [ PARTIAL I FULL ]    <expresión de tabla> )
  ```

  Por lo tanto, existen seis casos diferentes, dependiendo de (a) si se especifica o no la opción UNIQUE y (b) si se especifica o no la opción PARTIAL vs FULL (y de ser así, cuál es). Sin embargo, los detalles son complejos y están fuera del alcance de este libro. Para una mejor explicación vea la referencia [14.19].

  Condiciones EXISTS: vea el comentario de la referencia [18.6].

Expresiones escalares

- *"Literales":* La palabra reservada NULL puede ser usada como un tipo de representación literal de nulo (por ejemplo, en una instrucción INSERT). Sin embargo, observe que esta palabra reservada no puede aparecer en todos los contextos en los que aparecen las literales; ya que, como lo dice el estándar, "no existe <*literal*> para un valor nulo, aunque la palabra reservada NULL se usa en algunos lugares para indicar que es necesario un valor nulo" [4.22]. Por lo tanto no es posible, por ejemplo, especificar NULL en forma explícita como operando de una comparación simple. Es decir "WHERE X = NULL" no sería válido. (Por supuesto, la forma correcta es "WHERE X IS NULL").
- *COALESCE:* COALESCE es el análogo de SQL para nuestro operador IFJJNK (vea la sección 18.2).
- *Operadores de totales:* Los operadores de totales de SQL (como SUM, AVG, etcétera) *no* se comportan de acuerdo con las reglas de los operadores escalares que expliqué en la sec ción 18.2, sino que en vez de ello, simplemente ignoran cualquier nulo en sus argumentos; con excepción de COUNT(*), en donde los nulos son tratados como si fueran valores nor males. También, si el argumento de uno de estos operadores da como resultado un conjunto vacío, COUNT regresa cero y los demás operadores regresan nulo. (Como observamos en el capítulo 7, este último comportamiento es lógicamente incorrecto, pero es la forma en que está definido SQL.)
- *"Subconsultas escalares":* Si una expresión escalar es de hecho una expresión de tabla encerrada entre paréntesis —por ejemplo, (SELECT V.CIUDAD FROM V WHERE V.V# = 'VI')— entonces se requiere normalmente que esa expresión de tabla dé como resultado una tabla que contenga exactamente una columna y una sola fila. Entonces el valor de la expresión escalar se toma para que sea precisamente el único valor escalar que esté con tenido en esa tabla. Pero si la expresión de tabla da como resultado una tabla de una columna que no contenga ninguna fila, entonces SQL define el valor de la expresión escalar para que sea nulo.

Claves

La interacción entre los nulos y las claves en SQL puede resumirse de la siguiente forma:

- *Claves candidatas:* Hagamos que C sea una columna que es componente de alguna clave candidata *K* de alguna tabla base. Si *K* es una clave primaria, SQL no permitirá que C con tenga algún nulo (en otras palabras, hace cumplir la regla de integridad de la entidad). Sin embargo, si *K* no es una clave primaria, SQL permitirá que C contenga *cualquier cantidad* de nulos, por supuesto, junto con cualquier cantidad de valores no nulos distintos.
- *Claves externas:* Las reglas que definen lo que significa (en presencia de nulos) que un valor de clave externa dado concuerde con algún valor de la clave candidata correspondiente, son bastante complejas; aquí omitimos los detalles; sin embargo, como un mero comentario decimos que son básicamente los mismos que los de la condición MATCH (vea las secciones anteriores).

  Los nulos también tienen implicaciones para las acciones referenciales (como CAS-CADE, SET NULL, etcétera) especificadas en las cláusulas ON DELETE y ON UPDATE. (SET DEFAULT también es soportado, con la interpretación obvia.) Nuevamente, los de-talles son muy complejos y están fuera del alcance de este libro. Vea la referencia [4.19] para obtener detalles específicos.

SQL incrustado

Variables de indicador: Considere el siguiente ejemplo de un "SELECT individual" de SQL incrustado (una repetición del ejemplo del capítulo 4).

```
EXEC SQL SELECT STATUS, CIUDAD
         INTO  :CATEG0RIA, :CIUDAD
         FROM  V
         WHERE V# = :V#DAD0 ;
```

Supongamos que existe la posibilidad de que el valor de STATUS pueda ser nulo para algún proveedor. Entonces la instrucción anterior SELECT fallará si el STATUS seleccionado es nulo (SQLSTATE será establecido en el valor de excepción 22002). En general, si existe la posibilidad de que un valor a recuperar sea nulo, el usuario deberá especificar una *variable de indicador* adicional a la variable de destino ordinaria, como aquí:

```
EXEC SQL SELECT STATUS, CIUDAD
         INTO  ¡CATEGORÍA INDICATOR :INDCATEGORIA, :CIUDAD
         FROM  V
         WHERE V# • :V#DAD0 ; IF INDCATEGORIA = -1 THEN /*
STATUS fue nulo */ ... ; END IF ;
```

Si el valor a recuperar es nulo y se ha especificado una variable de indicador, entonces esa variable será establecida con el valor -1. El efecto sobre la variable de destino ordinaria depende de la implementación.

Ordenamiento: La cláusula ORDER BY se usa para ordenar las filas resultantes de la evaluación de la expresión de tabla en una definición de cursor. (Por supuesto, también puede ser usada en consultas interactivas.) Aquí surge la pregunta: ¿cuál es el ordenamiento relativo para dos valores escalares A y *B,* si *A* o *B* es nulo (o ambos)? La respuesta de SQL es la siguiente:

1. Para efectos de ordenamiento se considera que todos los nulos son iguales entre sí.
2. Para efectos de ordenamiento, se considera que todos los nulos son mayores que todos los valores no nulos *o bien,* menores que todos los valores no nulos (la determinación de qué posibilidad se aplica está definida por la implementación).

18.8 RESUMEN

Hemos explicado el problema de la **información faltante,** así como un enfoque para tratarlo que (aunque no es bueno) está de moda actualmente y el cual está basado en los **nulos y** la 3VL **(lógica de tres valores).** Enfatizamos el punto de que un nulo no es un valor, aunque es común hablar de él como si lo fuera (por ejemplo, diciendo que algún valor de atributo en particular, dentro de alguna tupia en particular "es nulo")- Cualquier comparación en la cual un operando es nulo da como resultado el "tercer valor de verdad", *desconocido* (abreviado *unk),* y ésta es la razón por la cual la lógica es de tres valores. También mencionamos que, al menos conceptualmente, puede haber muchas clases diferentes de nulos y presentamos a **UNK** como una abreviatura adecuada (y explícita) para la clase de "valor desconocido".

Luego exploramos las implicaciones de los UNK y la 3VL para las **expresiones lógicas,** los cuantificadores **EXISTS y FORALL,** las **expresiones computacionales y** los **operadores relaciónales** de producto, restricción, proyección, unión, intersección y diferencia, así como los operadores de actualización INSERT y UPDATE. Presentamos los operadores **IS_UNK** (que hace prueba para UNK), **IF_UNK** (que convierte UNK en un valor que no es UNK) y **MAYBE** (que convierte *unk* en *verdadero).* Explicamos la cuestión de los **duplicados** en presencia de UNK y señalamos también que UNK no es lo mismo que *unk.*

Después examinamos algunas consecuencias de las ideas anteriores. Primero, explicamos que **determinadas equivalencias se rompen** en la 3VL; equivalencias que son válidas en la 2VL pero no en la 3VL. Por consecuencia, es probable que tanto los usuarios como los optimizadores cometan **errores en la transformación de expresiones.** Y aunque no se cometan dichos errores, la 3VL sufre el problema muy serio ("conmocionante") de que **no concuerda con la realidad;** esto es, que los resultados que son correctos de acuerdo con la 3VL, a veces son incorrectos en la realidad.

Luego describimos las implicaciones de los nulos para las **claves primarias y** las **externas** (mencionamos, en particular, la regla de integridad de la **entidad** y la regla de integridad **referencial** revisada). Luego nos desviamos un poco para explicar la **junta externa.** No somos partidarios del soporte directo a esa operación (al menos no en la forma en que se comprende generalmente), ya que creemos que existen soluciones mejores al problema que trata de resolver la junta externa y en particular preferimos una solución que usa *atributos con valores de relación.* Brevemente mencionamos la posibilidad de otras operaciones "externas", en particular la **unión externa.**

Luego examinamos el **soporte de SQL** para las ideas anteriores. El manejo de la información faltante en SQL está basado ampliamente en la 3VL, pero incluye una gran cantidad de complicaciones adicionales, la mayoría fuera del alcance de este libro. Además, SQL presenta varias **fallas adicionales** por encima de las que ya son inherentes a la 3VL *por sí misma* [18.6], [18.10]. Lo que es más, estas fallas adicionales sirven como un **inhibidor** adicional **para la optimización** (como mencionamos al final del capítulo 17).

Cerramos con las siguientes observaciones:

- Apreciará que sólo hemos visto superficialmente los problemas que pueden presentarse con los nulos y la 3VL. Sin embargo, hemos tratado de abarcar lo suficiente para que quede claro que los "beneficios" del enfoque 3VL son un tanto dudosos.
- También debemos dejar claro que, aunque usted no esté convencido de los problemas de la 3VL *por sí misma,* es recomendable evitar las características correspondientes en SQL de bido a las "fallas adicionales" que mencionamos anteriormente.
- Nuestras recomendaciones para los usuarios de los DBMSs serían entonces ignorar por completo el soporte del fabricante para la 3VL y en su lugar, usar un esquema disciplinado de "valores especiales" (para permanecer, por lo tanto, firmemente en la lógica de dos va lores). Tal esquema se describe a detalle en la referencia [18.12].
- Por último, repetimos el siguiente punto fundamental de la sección 18.3: Hablando *muy* a la ligera, si el valor de un atributo dado dentro de una tupia dada dentro de una relación dada "es nulo", entonces esa posición de atributo en realidad no contiene nada... lo cual im plica que el "atributo" no es un atributo, la "tupia" no es una tupia, la "relación" no es una relación y los fundamentos para lo que estamos haciendo (independientemente de lo que pueda ser), ya no son teoría de relaciones matemáticas.

EJERCICIOS

18.1 Si A = 6, B = 5, C = 4 y D es UNK, establezca los valores de verdad de las siguientes expresiones:

a. `A = B 0R ( B > C AND A > D )`
b. `A > B AND ( B < C OR ISJJNK ( A - D ) )`
C. `A < C OR B < C OR NOT ( A = C )`
d. `B < D 0 R B = D 0 R B > D`
e. `MAYBE ( A > B AND B > C )`
f. `MAYBE ( ISJJNK ( D ) )`
g. `MAYBE ( ISJJNK ( A + B ) )`
h. `IFJJNK ( D, A ) > B AND IFUNK ( C, D ) < B`

18.2 Hagamos que la relación r contenga exactamente las siguientes tupias:

```
(    6,    5,    4 )
( UNK,    5,    4 )
(    6,  UNK,    4 )
( UNK,  UNK,    4 )
( UNK,  UNK,  UNK )
```

Al igual que en el cuerpo del capítulo, suponga que (a) los tres atributos, de izquierda a derecha, se llaman *A, B* y C, respectivamente y (b) todos los atributos son de tipo INTEGER. Si Ves una variable de alcance que se extiende sobre *r,* establezca los valores de verdad de las siguientes expresiones:

a. `EXISTS V ( V.B > 5 )`
b. `EXISTS V ( V.B > 2 AND V. C > 5 )`
C. `EXISTS V ( MAYBE ( V. C > 3 ) )`
d. `EXISTS V ( MAYBE ( ISJJNK ( V.C ) ) )`
e. `FORALL V ( V.A > 1 )`
f. `FORALL V ( V.B > 1 OR ISJJNK ( V.B ) )`
g. `FORALL V ( MAYBE ( V.A > V.B ) )`

18.3 Estrictamente hablando, el operador IS_UNK es innecesario, ¿por qué?

18.4 En la referencia [13.6], Codd propone versiones "tal vez" de algunos (no todos) operadores del álgebra relational. Por ejemplo, *maybe-restrict* difiere de la restricción normal en que regresa una relación cuyo cuerpo contiene solamente las tupias para las cuales la condición de restricción da como resultado *unk* en lugar de *verdadero.* Sin embargo, tales operadores son estrictamente innecesarios. ¿Por qué?

18.5 En la 2VL (lógica de dos valores) existen exactamente dos valores de verdad, *verdadero y falso.* Por consecuencia, existen exactamente cuatro operadores lógicos *monádicos* (de un solo operando) posibles; uno que transforma *verdadero y falso* en *verdadero,* otro que los transforma *en falso,* (

que transforma *verdadero en falso* y viceversa (éste es NOT, por supuesto) y otro que los deja sin cambio. Y hay exactamente 16 operadores diádicos (de dos operandos) posibles, como lo indica la siguiente tabla:

A	*B*															
v	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*v*	*f*	*f*	*f*	*f*	*f*	*f*	*f*	*f*
	f														*f*	*f*
v	*v*	*v*	*v*	*f*	*f*	*v*	*v*	*f*	*f*	*v*	*v*	*f*	*f*	*v*	*v*	*f*	*f*
v	*f*	*v*	*f*	*v*		*v*	*f*	*v*	*f*	*v*		*v*	*f*	*v*	*f*	*v*	*f*

Demuestre que en 2VL los cuatro operadores monádicos y los 16 operadores diádicos pueden ser formulados en términos de combinaciones convenientes de NOT y AND u OR (por lo tanto, no es necesario soportar explícitamente los 20 operadores).

18.6 ¿Cuántos operadores lógicos existen en la 3VL? ¿Y en la 4VL? Y en general, ¿en la «VL?

18.7 (Tomado de la referencia [18.5]). La figura 18.5 representa algunos valores de ejemplo para una ligera variación de la base de datos usual de proveedores y partes (la variación es que la varrel VP in cluye un nuevo atributo de *número de envío* ENVIO# y el atributo P# en esa varrel ahora tiene "UNKs permitidos"; la varrel P es irrelevante para el ejercicio y se ha omitido). Considere la consulta de cálculo relacional.

```
V WHERE NOT EXISTS VP ( VP.V# ■ V.V# AND
                        VP.P# • P# ( 'P2' ) )
```

(donde V y VP son variables de alcance implícitas). ¿Cuál de las siguientes (en caso de haberla) es una interpretación correcta de esta consulta?

a. Obtener los proveedores que no proporcionan P2.

b. Obtener los proveedores que no sabemos si proporcionan P2.

c. Obtener los proveedores que sabemos que no proporcionan P2.

d. Obtener los proveedores que sabemos que no proporcionan la parte P2 o que no sabemos si la proporcionan.

18.8 Diseñe un esquema de representación física para las tablas base de SQL, donde se permita que las columnas contengan nulos.

V

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres

VP

ENVIO#	V#	P#	CANT
ENVI01	V1	P1 P2	300
ENVI02	V2	UNK	200
ENVI03	V3		400

Figura 18.5 Una variación de proveedores y partes.

REFERENCIAS Y BIBLIOGRAFÍA

18.1 E. F. Codd y C. J. Date: "Much Ado about Nothing", en C. J. Date, *Relational Database Writ ings 1991-1994.* Reading, Mass.: Addison-Wesley (1995).

Probablemente Codd es el primer partidario de los nulos y la 3VL como base para manejar la información faltante. Este artículo contiene el texto de un debate sobre el tema entre Codd y un servidor. Incluye el siguiente comentario: "la administración de base de datos sería más fácil si no existieran los valores faltantes" (Codd).

18.2 Hugh Darwen: "Into the Unknown", en C. J. Date, *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Presenta varias cuestiones adicionales con relación a los nulos y la 3VL, de las cuales la siguiente es la más buscada: si (como mencioné en la respuesta al ejercicio 5.9 del capítulo 5) TABLE_DEE corresponde a *verdadero* y TABLE_DUM corresponde *a falso,* y TABLE_DEE y TABLE_DUM son las únicas relaciones posibles de grado cero, entonces **¿qué le corresponde** a ***unk?***

18.3 Hugh Darwen: "Outer Join with No Nulls and Fewer Tears", en C. J. Date y Hugh Darwen, *Re lational Database Writings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).

Propone una variación simple de la "junta externa" que no involucra nulos y que resuelve muchos de los problemas que debe resolver la junta externa.

18.4 C. J. Date: "The Outer Join", en *Relational Database: Selected Writings.* Reading, Mass.: Addison-Wesley (1986).

Analiza a profundidad el problema de la junta externa y presenta una propuesta para soportar la operación en un lenguaje como SQL.

18.5 C. J. Date: "NOT Is Not "Not"! (Notes on Three-Valued Logic and Related Matters)", en *Re lational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Supongamos que *X* es una variable de tipo BOOLEAN. Entonces *X* debe tener alguno de los valores *verdadero, falso* o *unk.* Por lo tanto, la instrucción *"X* no es *verdadero"* significa que el valor de *X* es *unk o falso.* Por el contrario, la instrucción *"X* es NOT *verdadero"* significa que el valor de *X es falso* (vea la tabla de verdad para NOT). Por lo tanto, el NOT de la 3VL no es el no del lenguaje natural... Este hecho ha ocasionado que varias personas (incluyendo a los diseñadores del SQL) se tropiecen y sin duda les volverá a pasar.

18.6 C. J. Date: "EXISTS Is Not "Exists"! (Some Logical Flaws in SQL)", en *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Muestra que el operador EXISTS de SQL no es lo mismo que el cuantificador existencial de la [1]VL, debido a que siempre da como resultado *verdadero o falso* y nunca *unk,* aunque *unk* es la respuesta lógicamente correcta.

18.7 C. J. Date: "Watch Out for Outer Join", en C. J. Date y Hugh Darwen, *Relational Database Writ ings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).

La sección 18.5 del presente capítulo menciona el hecho de que la junta externa sufre de varias "Propiedades Desagradables". Este artículo resume esas propiedades de la siguiente manera:

1. La junta @ externa no es una restricción del producto cartesiano.
2. La restricción no se distribuye sobre la junta @ externa.
3. *"A < B"* no es lo mismo que "A < *B* OR *A = B"* (en 3VL).

4. Los operadores de comparación-0 no son transitivos (en 3VL).
5. La junta natural externa no es una proyección de la equijunta externa.

El artículo considera lo que implica agregar soporte de la junta externa a la construcción SELECT-FROM-WHERE de SQL. Muestra que las propiedades desagradables anteriores implican que:

1. No funciona extender la cláusula WHERE.
2. No funciona aplicar AND entre juntas externas y restricciones.
3. No funciona la expresión de la condición de junta en la cláusula WHERE.
4. No es posible formular las juntas externas de más de dos relaciones sin expresiones anidadas.
5. No funciona extender la cláusula SELECT (sola).

El artículo también muestra que muchos productos existentes han caído en fallas con tales consideraciones.

18.8 C. J. Date: "Composite Foreign Keys and Nulls", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

¿Se debe permitir que los valores de claves externas compuestas sean completa o parcialmente nulos? Este artículo trata este asunto.

18.9 C. J. Date: "Three-Valued Logic and the Real World", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

18.10 C. J. Date: "Oh No Not Nulls Again", en C. J. Date y Hugh Darwen, *Relational Database Wri tings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Este artículo dice más de lo que usted probablemente quiera saber sobre los nulos.

18.11 C. J. Date: "A Note on the Logical Operators of SQL", en *Relational Database Writings 1991-1994*. Reading, Mass.: Addison-Wesley (1995).

Debido a que la 3VL tiene tres valores de verdad, *verdadero, falso* y *unk* (a los que aquí abreviamos como v,/y *u,* respectivamente), existen 3 * 3 * 3 = 27 operadores monádicos 3VL posibles, ya que cada una de las tres posibles entradas, v,/y *u* pueden transformarse hacia cada una de las tres posibles salidas v,/y *u.* Y hay 3^9 = 19,683 operadores diádicos 3VL posibles, como lo sugiere la siguiente tabla:

v	*u*	*f*
v/u/f	*v/u/f*	*v/u/f*
v/u/f	*v/u/f*	*v/u/f*
v/u/f	*v/u/f*	*v/u/f*

De hecho, en términos generales la lógica de *n* valores involucra *n* a la potencia *n* operadores monádicos y *n* a la potencia n^2 operadores diádicos:

	operadores monádicos	*operadores diádicos*
2VL	4 27	16 19,683
3VL	256	4,294,967,296
4VL		(n)**(n^2)
nVL		

Para cualquier nVL con $n > 2$ se presentan entonces las siguientes preguntas: ¿Cuál es un conjunto adecuado de operadores *primitivos?* (Por ejemplo, cualquiera de los conjuntos {NOT, AND} o (NOT, OR} es un conjunto primitivo adecuado para la 2VL.)

18.6 Vea el comentario de la referencia [18.11].

18.7 c. Para una explicación adicional vea la referencia [18.5]. *Ejercicio secundario:* dé una formu lación en cálculo relacional para la interpretación b.

18.8 Describimos brevemente la representación que se usa en DB2. En DB2 una columna que [aceptar nulos está representada físicamente en la base de datos almacenada mediante dos columnas, la propia columna de datos y una columna de indicador oculta (de 1 byte de ancho) que está guardada corno prefijo de la columna de datos actual. Un valor de columna indicadora de unos binarios indica que se debe ignorar el valor de la columna de datos correspondiente (es decir, tomarlo como nulo); un valor de columna indicadora de ceros binarios indica que el valor de la columna de datos corres pondiente se debe tomar como genuino. Pero la columna indicadora siempre está oculta (por supuesto) para el usuario.

CAPITULO 19

Herencia de tipo

19.1 INTRODUCCIÓN

En el capítulo 13 tratamos brevemente la idea de los subtipos y los supertipos —más específicamente, los subtipos y supertipos de *entidades*—, donde observamos que (por ejemplo) si algunos empleados son programadores, y todos los programadores son empleados, podemos considerar al tipo PROGRAMADOR como un *subtipo* del tipo de entidad EMPLEADO, y al tipo de entidad EMPLEADO como un *supertipo* del tipo de entidad PROGRAMADOR. Sin embargo, en ese capítulo también dijimos que un "tipo de entidad" no era un tipo en ningún sentido formal de ese término (ya que, en parte, el propio término "entidad" no está definido formalmente). En este capítulo analizaremos a profundidad los subtipos y supertipos, pero usaremos el término "tipo" en el sentido más formal y preciso del capítulo 5. Por lo tanto, comencemos definiendo cuidadosamente el término:

- Un **tipo** es **un** *conjunto de valores con nombre* (es decir, todos los valores posibles del tipo en cuestión), junto con un conjunto asociado de *operadores* que pueden ser aplicados a los valores y las variables del tipo en cuestión.

Además:

- Cualquier tipo dado puede ser definido por el sistema o por el usuario.
- Parte de la definición de cualquier tipo dado es una especificación del conjunto de todos los valores válidos de ese tipo (esta especificación es por supuesto la *restricción de tipo* aplicable, tal como la describimos en el capítulo 8).
- Estos valores pueden ser de una complejidad cualquiera.
- La representación real o física de dichos valores siempre está oculta ante el usuario; es decir, los *tipos* son distinguibles de sus *representaciones* (reales). Sin embargo, cada tipo tiene al menos una representación *posible* que es expuesta explícitamente ante el usuario por medio de los operadores THE_ convenientes (o algo equivalente lógico).
- Es posible operar sobre los valores y variables de un tipo determinado *únicamente* por medio de los operadores definidos para el tipo en cuestión.
- Además de los operadores THE_ que ya mencionamos, estos operadores incluyen:
  a. Al menos un operador *selector* (más precisamente, uno de estos operadores para cada representación posible expuesta), que permite que cada uno de los valores del tipo en cuestión sea "seleccionado" o especificado por medio de una llamada al selector apropiado;
  b. Un operador de *igualdad,* que permite que dos valores del tipo en cuestión sean probados para ver si en realidad son el mismo valor;

c. Un operador de *asignación,* que permite que un valor del tipo en cuestión sea asignado auna variable declarada del tipo en cuestión;

d. Determinados operadores *de prueba de tipo,* que explicaremos en la sección 19.6. *Nota:* Estos operadores pueden ser innecesarios en ausencia del soporte para la herencia.

Y a todo lo anterior, ahora añadimos:

- Algunos tipos son **subtipos** de otros **supertipos.** Si *B* es un subtipo de *A,* entonces todos los operadores y restricciones de tipo que se aplican a *A* también se aplican a *B* **(herencia),** pero *B* tiene operadores y restricciones de tipo propios que no se aplican a *A.*

Por ejemplo, supongamos que tenemos dos tipos, ELIPSE y CIRCULO, con las interpretaciones obvias. Entonces, podemos decir que el tipo CIRCULO es un subtipo del tipo ELIPSE (y el tipo ELIPSE es un supertipo del tipo CIRCULO), y con esto queremos decir que:

- Todo círculo es una elipse (es decir, el conjunto de todos los círculos es un subconjunto del conjunto de todas las elipses), pero lo contrario no es cierto.
- Por lo tanto, cada uno de los operadores que se aplican a las elipses en general, es aplica ble a los círculos en particular (debido a que los círculos *son* elipses), pero lo contrario no es cierto. Por ejemplo, el operador THE_CTRO ("el centro de") puede aplicarse a las elipses y por lo tanto a los círculos, pero el operador THE_R ("el radio de") puede aplicarse sola mente a los círculos.
- Además, cualquier restricción que se aplica a las elipses en general, se aplica también a los círculos en particular (de nuevo, debido a que los círculos *son* elipses), pero lo contrario no es cierto. Por ejemplo, si las elipses están sujetas a la restricción $a > b$ (en donde a y b son los semiejes mayor y menor, respectivamente), entonces esta misma restricción también debe ser satisfecha por los círculos. Por supuesto, para los círculos, a y b coinciden en el radio r y la restricción se satisface trivialmente; de hecho, la restricción $a = b$ es precisa mente una restricción que se aplica a los círculos en particular, pero no a las elipses en gene ral. *Nota:* A lo largo de este capítulo usamos el término "restricción" sin ningún calificativo para que signifique (específicamente) restricción de tipo. También usamos los términos "radio" y "semieje" para significar lo que debería ser llamado más adecuadamente como la *longitud* del radio o semieje correspondiente.

En resumen: en términos generales, el tipo CIRCULO hereda los operadores y las restricciones del tipo ELIPSE, pero también tiene operadores y restricciones propios que no se aplican al tipo ELIPSE. Por lo tanto, observe que el subtipo tiene un *subconjunto* de los valores pero un *superconjunío* de las propiedades; ¡un hecho que a veces puede causar confusión! *Nota:* Aquí, y a lo largo de este capítulo, usamos el término "propiedades" como una abreviatura adecuada para "operadores y restricciones".

¿Por qué herencia de tipo?

¿Por qué vale la pena investigar este tema? Parece haber, al menos, dos respuestas a esta pregunta:

- Las ideas de los subtipos y la herencia parecen surgir de forma natural en la realidad. Es decir, no es raro encontrar situaciones en las que todos los valores de un tipo dado tienen determinadas propiedades en común, mientras que algunos subconjuntos de esos valores

tienen propiedades adicionales especiales propias. Por lo tanto, los subtipos y la herencia parecen ser herramientas útiles para "modelar la realidad" (o *modelado semántico,* como lo llamamos en el capítulo 13).

- Segundo, si podemos reconocer esos patrones —es decir, los patrones de los subtipos y la herencia— y generamos información a partir de éstos en nuestro software de aplicación y de sistema, tal vez podamos lograr ciertas economías prácticas. Por ejemplo, un programa que funcione para las elipses también puede funcionar para los círculos, aunque haya sido escrito originalmente sin pensar para nada en los círculos (tal vez el tipo CIRCULO no había sido definido cuando se escribió el programa); y éste es el llamado beneficio de **reutilización del código.**

Sin embargo, a pesar de estas ventajas potenciales, observamos ahora que no parece haber ningún consenso sobre un **modelo** formal, riguroso y abstracto de la herencia de tipo. Para citar la referencia [19.10]:

> la idea básica de la herencia es muy simple... [y a pesar de] su papel central en los sistemas actuales... la herencia sigue siendo un mecanismo bastante controvertido... Todavía falta una vista completa de la herencia.

Las explicaciones de este capítulo están basadas en un modelo desarrollado por mí junto con Hugh Darwen, y está descrito en detalle en la referencia [3.3]. Por lo tanto, tenga presente que otros escritores y otros textos en ocasiones utilizan términos como "subtipo" y "herencia" de maneras diferentes a como nosotros lo hacemos. Queda advertido el lector.

Algunos puntos preliminares

Existen varios puntos preliminares que necesitamos aclarar antes de que podamos profundizar en una explicación adecuada de la herencia en sí. Estos puntos preliminares son el tema principal de esta subsección.

- *Los valores tienen tipo*

  Para repetir del capítulo 5: si v es un valor, entonces podemos pensar en v como si anduviera trayendo un tipo de bandera que anunciara "soy un entero" o "soy un número de proveedor" o "soy un círculo" (etcétera). Ahora bien, sin herencia todo valor es únicamente de un tipo. Pero con herencia un valor puede ser simultáneamente de varios tipos; por ejemplo, un valor dado puede ser de los tipos ELIPSE y CIRCULO al mismo tiempo.

- *Las variables tienen tipo*

  Toda variable tiene exactamente un tipo **declarado.** Por ejemplo, podemos declarar una variable de la siguiente forma:

  ```
  VAR E ELIPSE ;
  ```

  aquí el tipo declarado de la variable E es ELIPSE. Ahora, sin herencia todos los valores posibles de una variable dada son exactamente de un tipo, es decir el tipo declarado aplicable. Sin embargo, con herencia una variable dada puede tener un valor que es de varios tipos simultáneamente; por ejemplo, el valor actual de la variable E podría ser una elipse que es, de hecho, un círculo, y por lo tanto, ser de los tipos ELIPSE y CIRCULO al mismo tiempo.

Herencia simple contra múltiple

Hay dos grandes clases de la herencia de tipo: simple y múltiple. En términos generales, la **herencia simple** significa que cada subtipo tiene solamente un supertipo y sólo hereda propiedades de ese tipo; **herencia múltiple** significa que un subtipo puede tener cualquier cantidad de supertipos y hereda propiedades de todos ellos. Obviamente, el primero es un caso especial del segundo. Sin embargo, aun la herencia simple es bastante complicada (de manara sorprendente, en realidad); por lo tanto, en este capítulo limitamos nuestra atención a la herencia simple, y tomaremos el término *herencia* sin calificativos para dar a entender específicamente, herencia *simple.* Para un tratamiento detallado de ambos tipos de herencia (la múltiple y la simple), vea la referencia [3.3].

Herencia escalar, de tupia y de relación

De manera clara, la herencia tiene implicaciones para los valores que no son escalares y también para los escalares,* ya que a final de cuentas, esos valores que no son escalares están construidos con valores escalares. Por supuesto, tiene implicaciones en particular para los valores de tupia y de relación. Sin embargo, aun la herencia escalar es bastante complicada (de nuevo, de manera sorprendente); por lo tanto, en este capítulo limitamos nuestra atención a la herencia escalar y tomamos los términos *tipo, valor* y *variable* sin calificativos para dar a entender específicamente tipos, valores y variables **escalares.** Vea la referencia [3.3] para un tratamiento detallado sobre todos los tipos de herencia, de tupia y de relación, así como de la herencia escalar.

Herencia estructural contra la herencia de comportamiento

Recuerde que los valores escalares pueden tener una estructura o representación interna (física) de una complejidad cualquiera; por ejemplo, las elipses y los círculos pueden ser legítimamente considerados como valores escalares en circunstancias adecuadas (como ya sabemos), aunque su estructura interna pueda ser bastante compleja. Sin embargo, esa estructura interna siempre está *oculta ante el usuario.* De esto se deduce que cuando hablamos de herencia (al menos en lo que se refiere a nuestro modelo) *no* queremos decir herencia de estructura, ¡ya que desde el punto de vista del usuario no *hay* estructura a heredar! En otras palabras, estamos interesados en lo que a veces se llama herencia de **comportamiento** y no en la herencia **estructural** (donde "comportamiento" se refiere a los operadores; aunque le recordamos que al menos en nuestro modelo también se heredan las restricciones). *Nota:* Por supuesto, no excluimos la herencia estructural, sino que simplemente la vemos como un asunto de implementación y no es relevante para el modelo.

"Subtablas y supertablas "

Por ahora ya debe haber quedado claro que nuestro modelo de herencia se refiere a lo que en términos relaciónales podríamos llamar herencia de *dominio* (recuerde que los dominios y los tipos son lo mismo). Sin embargo, cuando nos preguntan sobre la posibilidad de la herencia en un contexto relacional, la mayoría de la gente supone instantáneamente que el punto

---

*Recuerde que un tipo escalar es aquel que no tiene componentes visibles por el usuario. No se confunda por el hecho de que los tipos escalares tienen *representaciones posibles* que a su vez tienen componentes visibles por el usuario, como explicamos en el capítulo 5; son componentes de las posibles representaciones y no del tipo, a pesar de que a veces, nos referimos a ellos —descuidadamente— como si en realidad fueran componentes del tipo.

que está a discusión es algún tipo de herencia de *tabla.* Por ejemplo, el SQL3 incluye soporte para algo a lo que llama "subtablas y supertablas", de acuerdo con lo cual, alguna tabla *B* puede heredar todas las columnas de alguna otra tabla *A* y luego añadir algunas más por su cuenta (vea el apéndice B). Sin embargo, nuestra posición es que la idea de "subtablas y supertablas" es un fenómeno totalmente aparte, que tal vez sea interesante —aunque somos escépticos [13.12]— pero no tiene nada que ver con la herencia de tipo en sí.

Un último comentario preliminar: en realidad, el tema de la herencia de tipo tiene que ver con los *datos en general;* no está limitado sólo a los datos de *bases de datos* en particular. Por lo tanto, la mayoría de los ejemplos de este capítulo están expresados (por razones de simplicidad) en términos de datos locales —variables ordinarias de programa, etcétera— en lugar de datos de la base de datos.

19.2 JERARQUÍAS DE TIPOS

Ahora presentamos un ejemplo que usaremos en el resto del capítulo. El ejemplo involucra un conjunto de tipos geométricos —FIGURA_PLANA, ELIPSE, CIRCULO, POLÍGONO, etcétera— arreglados en lo que se llama una **jerarquía de tipos** o, de manera más general, un **grafo de tipos** (vea la figura 19.1). Éste es un bosquejo de las definiciones en **Tutorial D** para algunos de los tipos geométricos (observe en particular las restricciones de tipo):

```
TYPE FIGURA_PLANA ... ;

TYPE ELIPSE POSSREP ( A LONGITUD, B LONGITUD, CTRO PUNTO )
     SUBTYPEJ3F ( FIGURA_PLANA ) CONSTRAINT ( THE_A (
     ELIPSE ) > THE_B ( ELIPSE ) ) ;

TYPE CIRCULO POSSREP ( R LONGITUD, CTRO PUNTO ) SUBTYPE_OF
     ( ELIPSE ) CONSTRAINT ( THEA ( CIRCULO ) = THE_B (
     CIRCULO ) ) ¡
```

Ahora el sistema sabe, por ejemplo, que CIRCULO es un subtipo de ELIPSE y por lo tanto, que los operadores y las restricciones que se aplican a las elipses en general, se aplican a los círculos en particular.

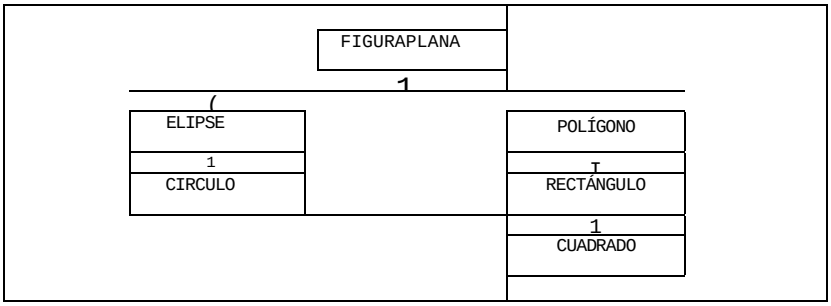


Figura 19.1 Un ejemplo de jerarquía de tipos.

Debemos comentar brevemente las especificaciones POSSREP para los tipos ELIPSE y CIRCULO. Por razones de simplicidad, estamos suponiendo básicamente que las elipses siempre están orientadas de forma tal que su eje mayor es horizontal y su eje menor es vertical; por lo tanto, las elipses podrían estar representadas por sus semiejes *a y b* (y su centro). Por el contrario, los círculos podrían estar representados por su radio *r* (y su centro). También estamos suponiendo, como lo hicimos en el capítulo 8, que el semieje mayor *a* de las elipses siempre es mayor o igual a su semieje menor *b* (es decir, son "bajas y gordas" y no "altas y flacas").

Este es un bosquejo de las definiciones para algunos de los operadores asociados con los tipos anteriores:

```
OPERATOR AREA ( E ELIPSE ) RETURNS ( AREA ) ;
   /* "área de" — observe que AREA es el nombre del •/
   /* operador en sí y del tipo del resultado */ ... ;
END OPERATOR ;

OPERATOR THE_A ( E ELIPSE ) RETURNS ( LONGITUD ) ;
   /* "el semieje a de" */ ... ; END
OPERATOR ;

OPERATOR THEB ( E ELIPSE ) RETURNS ( LONGITUD ) ;
   /* "el semieje b de" */ ... ; END
OPERATOR ;

OPERATOR THECTRO ( E ELIPSE ) RETURNS ( PUNTO ) ;
   /* "el centro de" */ ... ;
END OPERATOR ;

OPERATOR THER ( C CIRCULO ) RETURNS ( LONGITUD ) ;
   /* "el radio de" */ ... ;
END OPERATOR ;
```

Todos estos operadores, con excepción de THE_R, se aplican a valores de tipo ELIPSE y por lo tanto, necesariamente también a valores de tipo CIRCULO; por el contrario, THE_R sólo se aplica a valores de tipo CIRCULO.

Terminología

Por desgracia, hay más definiciones y términos que necesitamos presentar antes de continuar. Sin embargo, los conceptos son bastante directos.

1. Un supertipo de un supertipo es en sí mismo un supertipo; por ejemplo, POLÍGONO es un supertipo de CUADRADO.
2. Todo tipo es un supertipo de sí mismo; por ejemplo, ELIPSE es un supertipo de ELIPSE.
3. Si *A* es un supertipo de *B,* y *A* y *B* son distintos, entonces *A* es un supertipo **propio** de *B;* por ejemplo, POLÍGONO es un supertipo propio de CUADRADO.

Por supuesto, consideraciones similares se aplican a los subtipos. Por lo tanto:

4. Un subtipo de un subtipo es en sí mismo un subtipo; por ejemplo, CUADRADO es un sub tipo de POLÍGONO.
5. Todo tipo es un subtipo de sí mismo; por ejemplo, ELIPSE es un subtipo de ELIPSE.
6. Si *B* es un subtipo de *A,* y *B* y A son distintos, entonces *B* es un subtipo **propio** de *A;* por ejemplo, CUADRADO es un subtipo propio de POLÍGONO.

Además:

7. Si A es un supertipo de *B,* y no hay un tipo C que sea un subtipo propio de A y un supertipo propio de *B,* entonces *A* es un supertipo **inmediato** de *B* y *B* es un subtipo **inmediato** de *A.* Por ejemplo, RECTÁNGULO es un supertipo inmediato de CUADRADO y CUADRADO es un subtipo inmediato de RECTÁNGULO. (Por lo tanto, observe que en nuestra sintaxis de **Tutorial D** la palabra reservada SUBTYPE_OF significa específicamente, "subtipo *in mediato* de".)
8. Un tipo raíz es aquél que no tiene un supertipo propio; por ejemplo, FIGURA_PLANA es un *tipo miz. Nota:* Aquí no suponemos que haya un solo tipo raíz. Sin embargo, cuando hay dos o más, siempre podemos inventar alguna clase de tipo de "sistema" que sea un super tipo inmediato para todos ellos; y por lo tanto, no hay pérdida de generalidad al tomar sola mente uno.
9. Un tipo **hoja** es un tipo que no tiene subtipo propio; por ejemplo, CIRCULO es un tipo hoja. *Nota:* Esta definición está ligeramente simplificada, pero es adecuada para los propósitos actuales (necesita una ligera extensión para manejar adecuadamente la herencia múltiple [3.3]).
10. Todo subtipo propio tiene exactamente un supertipo inmediato. *Nota:* Aquí simplemente es tamos haciendo explícita nuestra suposición de que estamos manejando sólo la herencia simple. Como ya señalamos, dentro de la referencia [3.3] exploramos en detalle los efectos de relajar esta suposición.
11. Mientras (a) haya al menos un tipo y (b) no haya *ciclos* —es decir, no haya una secuencia de tipos *TI, T2, T3,..., Tn,* tal que 77 sea un subtipo inmediato de *T2, T2* sea un subtipo in mediato de 7!?, ..., y *Tn* sea un subtipo inmediato de *TI*— entonces al menos un tipo *debe* ser un tipo raíz. *Nota:* De hecho *no puede* haber ciclo alguno (¿por qué no?).

La suposición de tipos disjuntos

Hacemos una suposición adicional de simplificación, de la siguiente forma: si *TI* y *T2* son tipos raíz distintos o subtipos inmediatos distintos del mismo supertipo (lo que implica, en particular, que ninguno es subtipo del otro), entonces suponemos que son **disjuntos;** es decir, ningún valor es de ambos tipos *TI* y *T2.* Por ejemplo, ningún valor es al mismo tiempo elipse y polígono. Los siguientes puntos adicionales son consecuencias inmediatas de esta suposición:

12. Distintas jerarquías de tipo son disjuntas.
13. Distintos tipos hoja son disjuntos.
14. Cada valor tiene exactamente un tipo **más** específico. Por ejemplo, un valor dado puede ser "simplemente una elipse" y no un círculo, lo que significa que su tipo más específico es ELIPSE (en la realidad, algunas elipses no son círculos). De hecho, decir que el tipo más específico de algún valor v es *Tes* decir precisamente que el conjunto de tipos que posee v es el conjunto de todos los supertipos de T(un conjunto que incluye por supuesto a la propia 7).

Una razón por la que es necesaria la suposición de tipos disjuntos es que evita determinadas ambigüedades que podrían ocurrir. Supongamos que algún valor v pudiera ser de dos tipos, *TI y T2,* en donde ninguno fuera subtipo del otro. Supongamos además que se haya definido un

operador llamado *Op* para el tipo *TI* y otro operador con el mismo nombre *Op* para el tipo 72.* Entonces una llamada a *Op* con el argumento v sería ambigua.

Nota: La suposición de tipos disjuntos es razonable mientras limitemos nuestra atención a la herencia simple; aunque necesita ser relajada para la herencia múltiple. Para una explicación detallada, vea la referencia [3.3].

Una nota sobre la representación física

Aunque estamos tratando principalmente con un *modelo* de herencia (y no con asuntos de implementation) existen ciertos asuntos de implementación que usted debe comprender en cierta medida, para entender adecuadamente el concepto general de herencia; y en este momento vemos uno de ellos:

15. El hecho de que *B* sea un subtipo de *A* no implica que la representación actual (oculta) de los valores de *B* sea la misma que la de los valores de *AJ* Por ejemplo, las elipses pueden estar de hecho representadas por su centro y semiejes, mientras que (por el contrario) los círculos pueden estar representados por su centro y radio (aunque en general, no hay obligación de que la representación actual sea la misma que cualquiera de las posibles declaradas).

    Este punto llegará a ser importante en varias de las secciones que vienen a continuación.

19.3 EL POLIMORFISMO Y LA SUSTITUIBILIDAD

En esta sección consideramos dos conceptos cruciales, el *polimorfismo* y la *sustituibilidad,* que juntos proporcionan las bases para lograr el beneficio de la reutilización del código que mencionamos brevemente en la sección 19.1. Debemos decir inmediatamente que esos dos conceptos en realidad sólo son formas diferentes de ver lo mismo. Puesto que es así, comencemos dando un vistazo al polimorfismo.

Polimorfismo

La noción básica de herencia implica que si *T'* es un subtipo de *T,* entonces todos los operadores aplicados a los valores del tipo *T* también se aplican a los de tipo 7". Por ejemplo, si AREA(í) es válido (donde *e* es una elipse), entonces AREA(c) también debe ser válido (donde *c* es un círculo). Por lo tanto, observe que debemos ser muy cuidadosos en diferenciar entre los *parámetros* en

---

*En otras palabras, *Op* es un operador *polimórfico.* Lo que es más, el polimorfismo en cuestión podría ser de *sobrecarga* o de *inclusión.* Vea la sección 19.3 para mayores explicaciones.

[f]De hecho, no hay razón lógica por la cual todos los valores del *mismo* tipo deban tener la misma representación actual. Por ejemplo, algunos puntos pueden ser representados por coordenadas cartesianas y otros por coordenadas polares; algunas temperaturas pueden ser representadas en grados Celsius y otras en grados Fahrenheit; algunos enteros pueden ser representados en decimal y otros en binario; y así sucesivamente. (Por supuesto, en todos estos casos el sistema tendrá que saber cómo convertir las representaciones actuales para poder implementar adecuadamente las asignaciones, comparaciones, etcétera.)

términos de los cuales un operador dado está definido y sus tipos *declarados,* así como en diferenciar los *argumentos* correspondientes para una llamada dada a ese operador y sus tipos *actuales* (más específicos). Por ejemplo, el operador AREA está definido en términos de un parámetro de tipo declarado ELIPSE (vea la sección 19.2) pero el tipo actual —más específico— del argumento en la llamada a AREA(c) es **CIRCULO.**

Recuerde ahora que las elipses y los círculos (al menos como los definimos en la sección 19.2) tienen diferentes representaciones posibles:

```
TYPE ELIPSE POSSREP ( A LONGITUD, B LONGITUD, CTRO PUNTO ) ... ;

TYPE CIRCULO POSSREP ( R LONGITUD, CTRO PUNTO ) ... ;
```

Por lo tanto, es concebible que puedan existir dos versiones diferentes del operador AREA bajo la mesa; una que use la representación posible de ELIPSE y otra que use la de CIRCULO. Para repetir, es *concebible,* pero puede que no sea *necesario.* Por ejemplo, el código para ELIPSE podría ser como éste:

```
OPERATOR AREA ( E ELIPSE ) RETURNS ( AREA ) ;
   RETURN ( 3.14159 * THE_A ( E ) * THE_B ( E ) ) ;
END OPERATOR ;
```

(el área de la elipse es *trab).* Y este código funciona correctamente cuando es invocado con un círculo en lugar de una elipse más general ya que para un círculo, THE_A y THE_B regresan el radio *r.* Sin embargo, la persona responsable de la definición del tipo CIRCULO podría preferir, por muchas razones, implementar una versión distinta de AREA que sea específica para los círculos y que llame a THE_R en lugar de THE_A o THE_B. *Nota:* De hecho, quizá por razones de eficiencia podría ser necesario implementar las dos versiones del operador, aunque las representaciones posibles fueran las mismas. Por ejemplo, considere los polígonos y rectángulos. El algoritmo que calcula el área de un polígono general funcionará ciertamente para un rectángulo, pero para los rectángulos está disponible un algoritmo —multiplicar la anchura por la altura— más eficiente.

Sin embargo, observe que cuando el código de ELIPSE esté escrito en términos de la representación *actual* de ELIPSE (en vez de otra) y cuando la representación actual de los tipos ELIPSE y CIRCULO difiera, éste no funcionará para los círculos. Por lo general, la implementación de operadores en términos de las representaciones actuales, no es una buena idea. ¡Codifique a la defensiva!

De cualquier forma, si AREA *no* es reimplementada para el tipo CIRCULO, obtenemos una *reutilización del código* (para el código de la implementación de AREA). *Nota:* En la siguiente subsección encontraremos un tipo más importante de reutilización.

Por supuesto, desde el punto de vista del modelo, no hay diferencia sobre cuántas versiones de AREA existen bajo la mesa (en lo que se refiere al usuario, sólo existe por definición un operador AREA, el cual funciona para las elipses y por lo tanto también para los círculos). En otras palabras, desde el punto de vista del modelo, AREA es **polimórfico:** puede tomar argumentos de diferentes tipos en diferentes invocaciones. Por lo tanto, observe cuidadosamente que tal polimorfismo es una consecuencia lógica de la herencia; si tenemos herencia *debemos* tener polimorfismo, ya que de no ser así ¡no tenemos herencia!

Ahora, como ya se habrá dado cuenta, el polimorfismo en sí no es una idea nueva. Por ejemplo, SQL ya tiene operadores polimórficos ("=", "+", "II", y muchos otros), y de hecho ocurre

lo mismo con la mayoría de los lenguajes de programación. Algunos lenguajes permiten incluso que los usuarios definan sus propios operadores polimórficos; por ejemplo, PL/I proporciona dicha característica bajo el nombre "funciones GENERIC". Sin embargo, no hay herencia como tal involucrada en alguno de estos ejemplos; todos ellos son ejemplos de lo que en ocasiones se conoce como polimorfismo de *sobrecarga.* Por el contrario, al tipo de polimorfismo que presenta el operador AREA, se le conoce como polimorfismo de *inclusión,* sobre la base de que el vínculo que existe entre (digamos) círculos y elipses es básicamente el de una inclusión de conjuntos [19.3]. Por razones obvias, en el resto del capítulo tomamos el término "polimorfismo" sin calificativos para que signifique específicamente polimorfismo de inclusión (a menos que especifiquemos lo contrario).

Nota: Una manera útil de pensar sobre la diferencia entre el polimorfismo de sobrecarga y el de inclusión, es la siguiente:

- El polimorfismo de *sobrecarga* significa que existen varios operadores distintos con el mismo nombre (y el usuario no necesita saber que los operadores en cuestión son de hecho distintos —con semánticas distintas— aunque preferiblemente similares). Por ejemplo, el "+" está sobrecargado en la mayoría de los lenguajes (hay un operador "+" para la suma de enteros, otro operador "+" para la suma de números reales y así sucesivamente).
- El polimorfismo de *inclusión* significa que existe solamente un operador, con varias posi bles versiones diferentes de implementación (pero de hecho, el usuario no necesita saber si hay o no varias versiones de implementación; para repetir, para el usuario existe un solo operador).

Programación con polimorfismo

Considere el siguiente ejemplo. Suponga que necesitamos escribir un programa que despliegue un diagrama que esté compuesto de cuadrados, círculos, elipses, etcétera. Sin polimorfismo, el código se parecería al siguiente seudocódigo:

```
FOR EACH X IN DIAGRAMA
    CASE ;
       WHEN IS_CUADRADO ( X ) THEN CALL DESPLIEGACUADRADO ... ;
       WHEN IS_CIRCULO ( x ) THEN CALL DESPLIEGA_CIRCULO ... ;

    END CASE ;
```

(Estamos suponiendo la existencia de operadores IS_CUADRADO, IS_CIRCULO, etcétera, los cuales pueden ser usados para saber si un valor dado es del tipo especificado.) Por el contrario, con el polimorfismo, el código es mucho más simple y más conciso:

```
FOR EACH x IN DIAGRAMA CALL DESPLIEGA ( x ) ;
```

Explicación: Aquí, DESPLIEGA es un operador polimórfico. La versión de implementación de DESPLIEGA que funciona para valores de tipo *T,* será definida (generalmente) al definir el tipo *T* y se hará del conocimiento del sistema en ese momento. Y durante el tiempo de ejecución, cuando el sistema encuentre la invocación a DESPLIEGA con el argumento *x,* deberá determinar el tipo más específico de *x* y llamar a la versión de DESPLIEGA adecuada para ese tipo;

un proceso conocido como **enlace en tiempo de ejecución.*** En otras palabras, el polimorfismo en efecto significa que las expresiones e instrucciones CASE que tendrían que haber aparecido en el código fuente del usuario ahora están colocadas *bajo la mesa:* el sistema realiza efectivamente esas operaciones CASE en nombre del usuario.

Observe las implicaciones de lo anterior para el mantenimiento específico de programas. Por ejemplo, suponga que definimos un nuevo tipo TRIANGULO como otro subtipo inmediato de POLÍGONO y por lo tanto, el diagrama que va a ser desplegado puede ahora incluir también triángulos. Sin el polimorfismo, todo programa que contenga una expresión o instrucción CASE como la que mostramos antes, tendría que ser modificado para que incluyera código de la forma

```
WHEN IS_TRIANGULO ( X ) THEN CALL DESPLIEGA_TRIANGULO ... ;
```

Sin embargo, con el polimorfismo ya no son necesarias tales modificaciones al código fuente. Debido a ejemplos como el anterior, al polimorfismo se le caracteriza (a veces de manera un poco pintoresca) como que "permite que el código antiguo llame a código nuevo"; es decir, un programa *P* puede en efecto llamar a alguna versión de un operador que no existía al momento de escribir *P*. Por lo tanto, aquí tenemos otro ejemplo más importante de la *reutilización del código:* el mismo programa *P* podría ser utilizable sobre datos de un tipo Tque (para repetir), ni siquiera existía al escribir *P*.

Sustituibilidad

Como mencionamos anteriormente, el concepto de sustituibilidad es en realidad el concepto de polimorfismo visto desde un punto de vista ligeramente diferente. Hemos visto por ejemplo, que si AREA(e) es válido (donde *e* es una elipse) entonces AREA(c) también debe ser válido (donde *c* es un círculo). En otras palabras, donde quiera que el sistema espere una elipse, en su lugar podemos siempre sustituirla por un círculo. En términos más generales, donde quiera que el sistema espere un valor de tipo *T,* en su lugar podemos siempre sustituirlo con un valor de tipo 7" (donde 7" es un subtipo de 7); éste es el **principio de la sustituibilidad de valor.**

En particular, observe que este principio implica que si alguna relación r tiene un atributo *A* de tipo declarado ELIPSE, algunos de los valores de *A* en *r* pueden ser de tipo CIRCULO en lugar de sólo ser de tipo ELIPSE. De manera similar, si algún tipo *T* tiene una representación posible que involucre a un componente C de tipo declarado ELIPSE, entonces para algunos valores v de tipo *T,* la llamada al operador THE_C(v) podría regresar un valor de tipo CIRCULO en lugar de sólo uno de tipo ELIPSE.

Por último, observamos que debido a que en realidad se trata simplemente del polimorfismo puesto de otra forma, la sustituibilidad también es una consecuencia lógica de la herencia: si tenemos herencia *debemos* tener sustituibilidad, ya que de lo contrario no tenemos herencia.

---

*Por supuesto, el enlace en tiempo de ejecución es un asunto de implementation y no un asunto del modelo. Es otro de esos asuntos de implementation que usted tiene que apreciar (en cierta medida) para comprender adecuadamente el concepto general de la herencia.

19A VARIABLES Y ASIGNACIONES

Suponga que tenemos dos variables, E y C, de tipos declarados ELIPSE y CIRCULO, respectivamente:

```
VAR E ELIPSE ;
VAR C CIRCULO ;
```

Primero iniciamos C a algún círculo; digamos (sólo para ser concretos) el círculo con radio tres y centro en el origen:

```
C := CIRCULO ( LONGITUD ( 3.0 ), PUNTO ( 0.0, 0.0 ) ) ;
```

Aquí el lado derecho es una invocación al selector para el tipo CIRCULO. (Recuerde del capítulo 5 que para cada representación posible declarada hay un operador selector correspondiente, con el mismo nombre y con parámetros que corresponden a los componentes de la posible representación en cuestión. El propósito de un selector es permitir que el usuario especifique o "seleccione" un valor del tipo en cuestión, proporcionando un valor para cada componente de la representación posible en cuestión.)

Ahora considere la siguiente asignación:

```
E := C ;
```

Normalmente —es decir, en ausencia de subtipos y herencia— la operación de asignación requiere que la variable especificada del lado izquierdo y el valor indicado por la expresión del lado derecho, sean del mismo tipo (el mismo tipo *declarado*, en el caso de la variable). Sin embargo, *el principio de la sustituibilidad de valor* implica que donde quiera que el sistema espere un valor de tipo ELIPSE, siempre podremos sustituirlo por un valor de tipo CIRCULO y por lo tanto, la asignación es válida tal como se muestra (de hecho, la asignación es un operador polimórfico). Y el efecto es copiar el valor del círculo de la variable C a la variable E; en particular, el valor de la variable E después de la asignación, es de tipo CIRCULO y no simplemente de tipo ELIPSE. En otras palabras:

- **Los valores conservan su tipo más específico en la asignación a variables con tipo declarado menos específico.** En tales asignaciones *no* ocurre una conversión de tipo (en el ejemplo, el círculo *no* es convertido para que sea "sólo una elipse"). Observe que no queremos ninguna conversión, ya que esto ocasionaría que se perdiera el comportamiento más específico del valor; por ejemplo, en el caso que estamos considerando, podría sig nificar que después de la asignación ya no seríamos capaces de obtener el radio del valor círculo que está en la variable E. *Nota:* Vea la subsección "TREAT DOWN", que aparece más adelante en esta sección, para una explicación de lo que implica la obtención de ese radio.
- Deducimos que *la sustituibilidad implica que una variable de tipo declarado Tpuede tener un valor cuyo tipo más específico sea* ***cualquier subtipo*** *de T.* Por lo tanto, observe que aho ra debemos ser muy cuidadosos con la diferencia que existe entre el tipo *declarado* de una variable dada y el tipo *real* —es decir, el más específico— de (el valor actual de) esa varia ble. Regresaremos a este punto importante en la siguiente subsección.

Para continuar con el ejemplo, suponga que ahora tenemos otra variable A de tipo declarado AREA:

```
VAR A AREA ;
```

Considere la siguiente asignación:

```
A := AREA ( E ) ;
```

Lo que aquí pasa es lo siguiente:

- Primero, el sistema realiza una verificación de tipo, en tiempo de compilación, sobre la expresión AREA(E). Esa verificación es satisfactoria, debido a que E es del tipo declarado ELIPSE y el único parámetro para el operador AREA es también del tipo declarado ELIPSE (vea la sección 19.2).
- Segundo, al momento de la ejecución, el sistema descubre que el tipo más específico actual de E es CIRCULO y por lo tanto, llama a la versión de AREA que se aplica a los círculos (en otras palabras, realiza el proceso de enlace en tiempo de ejecución tal como mencionamos en la sección anterior).

Por supuesto, el hecho de que se invoque la versión de AREA para círculo y no a la de ELIPSE, no debe preocupar al usuario; para repetir, para el usuario sólo existe un operador AREA.

Variables escalares

Hemos visto que el valor actual v de una variable escalar *V* de tipo declarado *T*, puede tener cualquier subtipo de *T* como su tipo más específico. Deducimos que podemos (y lo hacemos) modelar a Vcomo un *triple ordenado* de la forma <*DT, MST,* v>, donde:

- *DT* es el tipo declarado para la variable *V.*
- *MST* es el tipo actual más específico de la variable *V.*
- v es un valor del tipo más específico *MST,* precisamente el valor actual de la variable *V.*

Usamos la notación *DT(V), MST(V)* y v(V) para referirnos a los componentes *DT, MST y* v (respectivamente) de este modelo de la variable escalar V. Observe que: (a) *MSTiV)* siempre es un subtipo —aunque no necesariamente un subtipo *propio*— de *DT(V);* (b) por lo general, *MST(V) y* v(V) cambian con el tiempo; (c) de hecho, *MST(V)* está implicado por v(V), debido a que todo valor es de exactamente un tipo más específico.

Este modelo de variable escalar es útil para concretar la semántica precisa de diversas operaciones, que incluyen —en particular— las operaciones de asignación. Sin embargo, antes de que podamos profundizar en este tema, debemos explicar que (por supuesto) las nociones del tipo declarado y del tipo más específico actual pueden extenderse en forma obvia para ser aplicadas a cualquier expresión escalar y también a las variables escalares en particular. Sea *X* una de esas expresiones. Entonces:

- *X* tiene un *tipo declarado, DT(X)* —más precisamente, el resultado de la evaluación de *X* tiene ese tipo—, derivado de manera obvia a partir de los tipos declarados para los operandos de *X* (que incluyen los tipos declarados para los resultados de cualquier invocación de operador contenida dentro de *X) y* que es *conocida en tiempo de compilación.*

- *X* también tiene un *tipo actual más específico, MST(X)* —para ser más precisos, el resultado de la evaluación de *X* tiene ese tipo— derivado de manera obvia, a partir de los valores actuales de los operandos de *X* (incluyendo los valores actuales de los resultados de cualquier invocación a operador que esté contenida dentro de *X)* y el cual generalmente *no es conocido sino hasta el tiempo de ejecución.*

Ahora podemos explicar adecuadamente la asignación. Considere la asignación

```
V := X ;
```

(donde *V* es una variable escalar y X es una expresión escalar). *DT(X)* debe ser un subtipo de *DT(V),* ya que en caso contrario, la asignación no será válida (ésta es una verificación en tiempo de compilación). Si la asignación es válida su efecto será hacer que *MST(V)* sea igual a *MST(X)* y v(V) igual a *v(X).*

Por cierto, observe que si el tipo actual más específico de la variable *V* es *T,* entonces cada supertipo propio de tipo *T* también es un "tipo actual" de la variable *V.* Por ejemplo, si la variable E (de tipo declarado ELIPSE) tiene un valor actual de tipo más específico CIRCULO, entonces CIRCULO, ELIPSE y FIGURA_PLANA son todos "tipos actuales" de E. Sin embargo, por lo general la frase "tipo actual de X" se toma (al menos informalmente) para que signifique específicamente *MST(X).*

Repaso de la sustituibilidad

Considere la siguiente definición de operador:

```
OPERATOR COPIA ( E ELIPSE ) RETURNS ( ELIPSE ) ;
   RETURN ( E ) ;
END OPERATOR ;
```

Debido a la sustituibilidad, el operador COPIA puede ser obviamente invocado con un argumento de tipo más específico —ya sea ELIPSE o CIRCULO— y cualquiera que sea, regresará claramente un resultado de ese mismo tipo más específico. Deducimos que la noción de sustituibilidad tiene la implicación adicional de que *si el operador Op está definido para que tenga un resultado de tipo declarado T, entonces el resultado real de una invocación a Op puede ser de cualquier subtipo de T* (en general). En otras palabras, así como (a) una referencia a una variable de tipo declarado *T* puede denotar de hecho un valor de cualquier subtipo de T(en general), así (b) una invocación de un operador con un tipo de resultado declarado *T* puede de hecho regresar un valor de cualquier subtipo de *T* (otra vez, en general).

TREAT DOWN

Nuevamente, aquí está el ejemplo del inicio de esta sección:

```
VAR E ELIPSE ;
VAR C CIRCULO ;

C := CIRCULO ( LONGITUD ( 3.0 ), PUNTO ( 0.0, 0.0 ) ) ;
E := C ;
```

MST(E) es ahora CIRCULO. Supongamos ahora que queremos obtener el radio del círculo en cuestión y asignarlo a alguna variable L. Podríamos intentar lo siguiente:

```
VAR L LONGITUD ;
L : = THER ( E ) ; /* I ti error üe tipo en tiempo de compilación III */
```

Sin embargo, como lo indica el comentario, este código falla por un error de tipo en tiempo de compilación. Para ser más específicos, falla debido a que el operador THE_R ("el radio de") del lado derecho de la asignación requiere un argumento de tipo CIRCULO, y el tipo declarado del argumento E es ELIPSE y no CIRCULO. *Nota:* Si no se hiciera la revisión del tipo en tiempo de compilación, podríamos obtener un error de tipo en *tiempo de ejecución* —que es peor— cuando el valor actual de E en tiempo de ejecución fuera sólo una elipse y no un círculo. Por supuesto, en el caso que estamos viendo sabemos que el valor en tiempo de ejecución será un círculo; el problema es que nosotros sabemos esto, pero el compilador no.

Para manejar dichos problemas presentamos un nuevo operador, al que llamamos informalmente como *TREAT DOWN.* La forma correcta para obtener el radio del círculo en este ejemplo es la siguiente:

```
L := THE_R ( TREAT_DOWN_AS_CIRCULO ( E ) ) ;
```

La expresión TREAT_DOWN_AS_CIRCULO(E) está definida para que tenga un tipo declarado igual a CIRCULO, para que ahora la verificación de tipo en tiempo de compilación sea válida. Entonces, en tiempo de ejecución:

- Si el valor actual de E ya es de tipo CIRCULO, la expresión general regresa correctamente el radio de ese círculo. Para ser más precisos, la llamada a TREAT DOWN produce un re sultado —digamos Z— con (a) el tipo declarado *DT(Z)* igual a CIRCULO, debido a la es pecificación "..._AS_CIRCULO"; (b) el tipo actual más específico de *MST(Z)* igual a MST(E), que también es CIRCULO en el ejemplo; (c) el valor actual de v(Z) igual a v(E); entonces (d) se evalúa la expresión "THE_R(Z)" para dar el radio deseado (el cual puede ser asignado posteriormente para L).
- Sin embargo, si el valor actual de E es sólo de tipo ELIPSE, y no CIRCULO, entonces TREAT DOWN falla por un error de tipo en *tiempo de ejecución.*

El propósito general de TREAT DOWN es asegurar que los errores de tipo en tiempo de ejecución sólo puedan suceder en el contexto de una invocación a TREAT DOWN.

Nota: Supongamos que CIRCULO tiene a su vez un subtipo propio, digamos CIRCULO_O (donde un "círculo-O" es un círculo que está centrado en el origen):

```
TYPE CIRCULOJ) POSSREP ( R LONGITUD ) SUBTYPE_OF ( CIRCULO )
   CONSTRAINT ( THECTRO ( CIRCULOJ) ) = PUNTO ( 0.0, 0.0 ) ) ;
```

Entonces en algún momento dado, el valor actual de la variable E puede ser del tipo más específico CIRCULO_O, en lugar de ser simplemente CIRCULO. Si esto es así, la invocación a TREAT DOWN

```
TREAT_DOWN_AS_CIRCULO ( E )
```

será satisfactoria y producirá un resultado —digamos Z— con (a) *DT(Z)* igual a CIRCULO, debido a la especificación "..._AS_CIRCULO"; (b) *MST(Z)* igual a CIRCULO_O, ya que CIRCULO_O es el tipo más específico de E; y (c) v(Z) igual a v(E). En otras palabras (en general), TREAT DOWN siempre deja solo al tipo más específico y nunca lo "empuja hacia arriba" para hacerlo menos específico de lo que era antes.

Para una referencia futura, aquí tenemos una declaración formal de la semántica para la llamada al operador TREAT_D0WN_AS_7\X); donde *X* es alguna expresión escalar. En primer lugar, *T* debe ser un subtipo de *DT(X)* (ésta es una verificación en tiempo de compilación). Segundo, *MST(X)* debe ser un subtipo de *T* (ésta es una verificación en tiempo de ejecución). Si suponemos que estas condiciones son satisfechas, la invocación regresa un resultado Z con *DT(Z)* igual a *T, MST(Z)* igual a *MST(X)* y v(Z) igual a v(X). *Nota:* La referencia [3.3] también define una forma generalizada de TREAT DOWN que permite que un operando sea "tratado hacia abajo" para el tipo de otro, en lugar de ser tratado para algún tipo nombrado específicamente.

19.5 ESPECIALIZACION POR RESTRICCIÓN

Considere el siguiente ejemplo de una llamada al selector para el tipo ELIPSE:

```
ELIPSE ( LONGITUD ( 5.0 ), LONGITUD ( 5.0 ), PUNTO (...))
```

Esta expresión regresa una elipse con semiejes iguales. Pero en la realidad, una elipse con semiejes iguales es de hecho un círculo; entonces, ¿esta expresión regresa un resultado del tipo más específico CIRCULO en lugar del tipo más específico ELIPSE?

En la literatura se han presentado muchas controversias acaloradas (y de hecho, todavía se presentan) sobre cuestiones como ésta. En nuestro propio modelo decidimos, después de haberlo pensado cuidadosamente, que es mejor insistir en que la expresión sí regresa un resultado de tipo más específico CIRCULO. En términos más generales, si el tipo *V* es un subtipo del tipo *T,* y una invocación al selector para el tipo *T* regresa un valor que satisface las restricciones de tipo para el tipo *T',* entonces (en nuestro modelo) el resultado de esa invocación al selector es de tipo 7".* *Nota:* Debe considerar que algunas de las implementaciones comerciales actuales (si es que las hay) se comportan de esta forma en la práctica, aunque vemos este hecho como una falla por parte de esos sistemas. La referencia [3.3] muestra que debido a esa falla, esos sistemas están forzados a soportar "círculos no circulares", "cuadrados no cuadrados" y cosas similares sin sentido; una crítica que no se aplica a nuestro enfoque.

De lo anterior deducimos que (al menos en nuestro modelo) ningún valor del tipo más especifico ELIPSE tendrá alguna vez *a = b;* en otras palabras, los valores del tipo más específico ELIPSE corresponden precisamente a las elipses reales que no son círculos. Por el contrario, los valores del tipo más específico ELIPSE corresponden —en otros modelos de herencia— a las elipses reales que *pueden o no* ser círculos. Por lo tanto, sentimos que nuestro modelo es un poco más aceptable como "un modelo de la realidad".

A la idea de que (por ejemplo) una elipse con *a = b* debe ser de tipo CIRCULO, se le conoce como **especialización por restricción** [3.3]; aunque debemos prevenirle que otros escritores usan este término para que signifique algo completamente diferente (vea por ejemplo, las referencias [19.7] y [19.11]).

---

*La referencia [3.3] sugiere que este efecto puede lograrse por medio de una cláusula SPECIALIZE en la definición del tipo *T.* Sin embargo, hemos llegado a la conclusión de que no se necesita ninguna sintaxis especial para lograr el efecto que deseamos.

Revisión de las seudovariables THE_

Recuerde del capítulo 5 que las seudovariables THE_ proporcionan una forma de actualizar un componente de una variable, dejando a los demás componentes sin cambio (aquí "componentes" se refiere, por supuesto, a los componentes de alguna representación *posible* y no necesariamente de una representación *real).* Por ejemplo, sea la variable E de tipo declarado ELIPSE, y sea el valor actual de E una elipse con (digamos) *a* igual a cinco y *b* igual a tres. Entonces la asignación

```
THEB ( E ) := LONGITUD ( 4.0 ) ;
```

actualiza el semieje *b* de E a cuatro, sin cambiar el semieje *a* ni el centro.

Ahora, como observamos en el capítulo 8, sección 8.2, las seudovariables THE_ son lógicamente innecesarias; ya que en realidad sólo son abreviaturas. Por ejemplo, la asignación que acabamos de mostrar, la cual usa una seudovariable THE_, es una abreviatura para la siguiente que no la usa:*

```
E := ELIPSE ( THE_A ( E ), LONGITUD ( 4.0 ), THE_CTRO ( E ) ) ;
```

Por lo tanto, considere la siguiente asignación:

```
THE_B ( E ) := LONGITUD ( 5.0 ) ;
```

Por definición, esta asignación es equivalente a la siguiente:

```
E := ELIPSE ( THE_A ( E ), LONGITUD ( 5.0 ), THE_CTRO ( E ) ) ;
```

Por lo tanto, entra en juego la especialización por restricción (debido a que la expresión del lado derecho regresa una elipse con *a = b)* y el efecto neto es que después de la asignación, *MST(E)* es CIRCULO y no ELIPSE.

Entonces considere la asignación:

```
THEB ( E ) := LONGITUD ( 4.0 ) ;
```

Ahora E contiene una elipse con *a* igual a cinco y *b* igual a cuatro (como era antes) y *MST(E)* se convierte nuevamente en ELIPSE; un efecto al que nos referimos como **generalización** por restricción.

Nota: Supongamos (como hicimos casi al final de la sección 19.4) que el tipo CIRCULO tiene un subtipo propio CIRCULO_O (donde "círculo-O" es un círculo con centro en el origen):

```
TYPE CIRCULOO POSSREP ( R LONGITUD ) SUBTYPE_OF ( CIRCULO )
   CONSTRAINT ( THE_CTR0 ( CIRCULO_O ) ■ PUNTO ( 0.0, 0.0 ) ) ¡
```

Entonces el valor actual de la variable E puede en un momento dado ser del tipo más específico CIRCULO_O, en lugar de ser simplemente CIRCULO. Supongamos que así es y consideremos la siguiente secuencia de asignaciones:[1]

---

*De paso, hacemos notar que TREAT DOWN también puede ser usado como una seudovariable [3.3]; aunque de nuevo es en efecto sólo una abreviatura.

*Como dijimos en el capítulo 8, la referencia [3.3] propone una forma *múltiple* de asignación que permitiría que la secuencia de asignaciones sea ejecutada como una operación única.

```
THEA ( E ) := LONGITUD ( 7.0 ) ;
THE_B ( E ) := LONGITUD ( 7.0 ) ;
```

Después de la primera de estas asignaciones, E contendrá "sólo una elipse", gracias a la generalización por restricción. Sin embargo, después de la segunda, contendrá nuevamente un círculo, ¿pero será específicamente un Circulo_O o "sólo un círculo"? Obviamente quisiéramos que fuera específicamente un Circulo_O. Y de hecho así es, ya que satisface precisamente la restricción para el tipo CIRCULO_O (incluyendo la restricción heredada por ese tipo desde el tipo CIRCULO).

Cambio de tipos hacia los lados

De nuevo, sea E una variable de tipo declarado ELIPSE. Hemos visto la manera de cambiar el tipo de E "hacia abajo" (por ejemplo, si su tipo actual más específico es ELIPSE, hemos visto la manera de actualizarlo para que su tipo actual más específico se convierta en CIRCULO); también hemos visto la manera de cambiar el tipo de E "hacia arriba" (por ejemplo, si su tipo actual más específico es CIRCULO, hemos visto la manera de actualizarlo a fin de que su tipo actual más específico se convierta en ELIPSE). ¿Pero qué hay acerca de cambiar el tipo "hacia los lados"? Supongamos que extendemos nuestro ejemplo en forma tal que el tipo ELIPSE tenga dos subtipos inmediatos, CIRCULO y NOCIRCULO (con sus significados obvios).* Sin entrar en demasiados detalles, debe quedar claro que:

- Si el valor actual de E es del tipo CIRCULO (y por lo tanto, *a = b),* la actualización de E de tal forma que *a > b* causará que *MST(E)* se convierta en NOCIRCULO;
- Si el valor actual de E es de tipo NOCIRCULO (y por lo tanto, *a > b),* la actualización de E de tal forma que *a = b* causará que *MST(E)* se convierta en CIRCULO.

Por lo tanto, la especialización por restricción también se encarga de los cambios de tipo "hacia los lados". *Nota:* En caso de que se lo pregunte, la actualización de E para que *a < b* es imposible (viola la restricción del tipo ELIPSE).

19.6 COMPARACIONES

Supongamos que tenemos nuestras dos variables usuales E y C de tipos declarados ELIPSE y CIRCULO, respectivamente, y supongamos que asignamos el valor actual de C a E:

```
E := C ;
```

Entonces, es claramente obvio que si realizamos la comparación de igualdad

```
E = C
```

debemos obtener el resultado *verdadero;* y de hecho así es. La regla general es la siguiente. Considere la comparación *X = Y* (donde *X* y *Y* son expresiones escalares). El tipo declarado *DT(X)* debe ser un subtipo del tipo declarado *DT(Y)* o al revés, ya que en caso contrario la comparación

---

*Dicho sea de paso, ELIPSE se convierte ahora en un tipo ficticio *(dummy);* vea la sección 19.7.

no será válida (ésta es una verificación en tiempo de compilación). Si la comparación es válida, el efecto será regresar *verdadero* si el tipo más específico *MST(X)* es igual al tipo más específico *MST(Y)* y el valor *v(X)* es igual al valor *v(Y);* y regresará *falso* en caso contrario. En particular, observe que dos valores no tienen la posibilidad de "compararse por igual" si sus tipos más específicos son diferentes.

Efectos en el álgebra relacional

Las comparaciones por igualdad están involucradas, implícita o explícitamente, en muchas de las operaciones del álgebra relacional. Y cuando están involucrados supertipos y subtipos, resulta que algunas de esas operaciones presentan un comportamiento que podría ser visto (al menos a primera vista) como un poco contraintuitivo. Considere las relaciones RX y RY que muestra la figura 19.2. Observe que el único atributo A de RX es de tipo declarado ELIPSE, y su contraparte A en RY es de tipo declarado CIRCULO. Adoptamos la convención en la figura de que los valores de la forma E¡ son elipses que no son círculos, mientras que los valores de la forma C¡ son círculos. Mostramos los tipos más específicos en minúsculas y con cursivas.

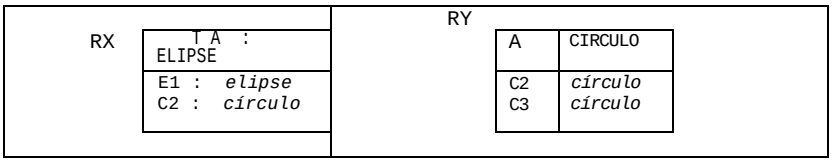


Figura 19.2 Las relaciones RX y RY.

Ahora considere la junta de RX y RY, digamos RJ (vea la figura 19.3). De manera clara, todo valor de A en RJ será necesariamente de tipo CIRCULO (debido a que cualquier valor de A en RX, cuyo tipo más específico sea simplemente ELIPSE, no tiene la posibilidad de "compararse por igual" con ningún valor de A en RY). Por lo tanto, podríamos pensar que el tipo declarado del atributo A en RJ debe ser CIRCULO y no ELIPSE. Pero considere lo siguiente:

- Puesto que RX y RY tienen a A como su único atributo, RX JOIN RY se reduce a RX INTERSECT RY. Por lo tanto, en tales circunstancias, la regla que se refiere al tipo declarado del atributo del resultado de JOIN, debe ser obviamente reducida a la regla equivalente para INTERSECT.

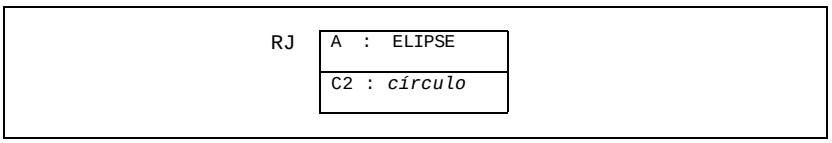


Figura 19.3 La junta RJ de las relaciones RX y RY.

- RX INTERSECT RY es a su vez lógicamente equivalente a RX MINUS (RX MINUS RY). Sea aquí RZ el resultado del segundo operando (es decir, RX MINUS RY). Entonces debe quedar claro que:
  a. En general, RZ incluirá algunos valores de A para tipo más específico ELIPSE, y por lo tanto, el tipo declarado del atributo A en RZ debe ser ELIPSE.
  b. Por lo tanto, la expresión original se reduce a RX MINUS RZ, donde el tipo declarado del atributo A en RX y RZ, es ELIPSE y por lo tanto, produce un resultado final en el cual el tipo declarado del atributo A debe ser obviamente otra vez ELIPSE .
- Deducimos que el tipo declarado del atributo del resultado para INTERSECT (y por lo tanto, también para JOIN) debe ser ELIPSE y no CIRCULO; aun cuando (para repetir) ¡todo *valor* de ese atributo debe ser de tipo CIRCULO!

Ahora veamos el operador relacional de diferencia, MINUS. Primero consideremos RX MINUS RY. Debe quedar claro que algunos valores de A en el resultado de esta operación, serán de tipo ELIPSE y no CIRCULO; *y* por \o tanto, el upo declarado de A en ese resultado *debe* ser de tipo ELIPSE. Pero, ¿qué hay acerca de RY MINUS RX? De manera clara, cada uno de los valores de A en el resultado de esta última operación, será de tipo CIRCULO y por lo tanto, de nuevo podríamos pensar que el tipo declarado de A en ese resultado debe ser CIRCULO y no ELIPSE. Sin embargo, observe que RX INTERSECT RY es lógicamente equivalente no sólo a RX MINUS (RX MINUS RY), como ya explicamos, sino también a RY MINUS (RY MINUS RX); debido a esto, es fácil ver que especificar que el tipo declarado de A en el resultado de RY MINUS RX es CIRCULO, da lugar a una contradicción. Deducimos que el tipo declarado del atributo del resultado para MINUS, también debe ser ELIPSE y no CIRCULO; incluso en el caso de RY MINUS RX, donde cada uno de los *valores* de ese atributo debe ser de hecho del tipo CIRCULO.

Por último, considere RX UNION RY. En este caso debe ser obvio que, en general, el resultado incluirá algunos valores de A de tipo más específico ELIPSE; y por lo tanto, el tipo declarado del atributo A en ese resultado debe ser necesariamente ELIPSE. Entonces, el tipo declarado del atributo del resultado para UNION también debe ser ELIPSE (pero este caso particular —a diferencia de JOIN, INTERSECT y MINUS— difícilmente puede ser visto como contraintuitivo).

Entonces ésta es la regla general:

- Sean *rx* y *ry* relaciones con un atributo común A, y sean *DT(Ax)* y *DT(Ay)* los tipos declara dos de A en rx y *ry*, respectivamente. Considere la junta de *rx* y *ry* (necesariamente sobre *A*, al menos en parte). *DT(Ax)* debe ser un subtipo de *DT(Ay)* o *viceversa*, ya que de no ser así la junta no es válida (ésta es una verificación en tiempo de compilación). Si la junta es válida damos por hecho (sin perder la generalidad) que *DT{Ay)* es un subtipo de *DT(Ax)*. Entonces, el tipo declarado de *A* en el resultado es *DT(Ax)*.
- Consideraciones similares se aplican a unión, intersección y diferencia. En cada caso, (a) los atributos correspondientes de los operandos deben ser tales que el tipo declarado de uno sea un subtipo del tipo declarado del otro, y (b) el tipo declarado del atributo correspondiente en el resultado es el menos específico de los dos (donde por **menos es pecífico** de los dos tipos *T* y 7" —uno de los cuales es subtipo del otro— nos referimos al que sea supertipo).

Prueba de tipo

En una sección anterior mostramos un fragmento de código que utilizó operadores de la forma IS_CUADRADO, IS_CIRCULO, etcétera, para probar si un valor especificado era de algún tipo especificado. Es tiempo de dar una vistazo más cercano a estos operadores. En primer lugar, suponemos que la definición de un tipo dado *T* ocasiona la definición automática de un operador que da un valor de verdad, de la forma

is_r (*x*)

donde *X* es una expresión escalar tal que *DT(X)* es un supertipo de *T* (ésta es una verificación en tiempo de compilación). La expresión general da *verdadero* cuando *X* es de tipo *T, y falso* en caso contrario. Observe que (para repetir) el tipo declarado del argumento *X* especificado debe ser un supertipo del tipo *T* especificado. Entonces, por ejemplo, si C es una variable de tipo declarado CIRCULO, la expresión

```
IS_CUADRADO { C )
```

no es válida (falla por un error de tipo en tiempo de compilación). Por otro lado, las dos expresiones siguientes son válidas y ambas dan *verdadero:*

```
IS_CIRCULO ( C )
IS_ELIPSE ( C )
```

Y si E es una variable de tipo declarado ELIPSE, pero su tipo más específico actual es algún subtipo de CIRCULO, la expresión

```
IS_CIRCULO ( E )
```

también da *verdadero**

También suponemos que la definición de un tipo Tdado, ocasiona la definición automática de un operador de la forma

```
ISJKS_T ( x )
```

donde *X* es una expresión escalar y *DT(X)* es un supertipo de *T* (de nuevo, ésta es una verificación en tiempo de compilación). La expresión general da *verdadero* si *X* es de tipo más específico *T, y falso* en caso contrario. *Nota:* Puede serle de utilidad observar que mientras, por ejemplo, el operador IS_ELIPSE se transforma en lenguaje natural a "es una elipse", el operador IS_MS_ELIPSE se transforma en lenguaje natural a "es *más específicamente* una elipse".

Éste es un ejemplo que involucra rectángulos, cuadrados e IS_MS_RECTANGULO.

```
VAR R RECTÁNGULO ;

IF IS_MS_RECTANGULO ( R )
   THEN CALL ROTAR ( R ) ;
END IF ;
```

---

*La referencia [3.3] define formas generalizadas de todos los operadores de "pruebas de tipos" presentados en esta subsección; por ejemplo, una forma generalizada de IS_T que prueba si un operando es del mismo tipo que otro, en vez de probar simplemente si es de algún tipo nombrado explícitamente.

Lo que intuimos tras este ejemplo es que (a) ROTAR es un operador que gira su argumento rectángulo 90° alrededor de su centro y (b) no tiene caso girar un rectángulo de esta forma cuando el rectángulo en cuestión es un cuadrado.

La prueba de tipo también tiene implicaciones con los operadores relaciónales. Considere el siguiente ejemplo. Sea R una varrel que tiene un atributo A de tipo declarado ELIPSE; suponga ahora que queremos obtener las tupias de R donde el valor de A sea de hecho un círculo y cuyo radio sea mayor que dos. Podemos intentar lo siguiente:

```
R WHERE THE_R ( A ) > LONGITUD ( 2.0 )
```

Sin embargo, esta expresión fallará por un error de tipo en tiempo de compilación, ya que THE_R requiere un argumento de tipo CIRCULO y el tipo declarado de A es ELIPSE, no CIRCULO. (Por supuesto, si no se realizara la verificación de tipo en tiempo de compilación, obtendríamos un error de tipo en *tiempo de ejecución* tan pronto como encontráramos una tupia en la cual el valor de A fuera simplemente una elipse y no un círculo.)

De manera clara, lo que debemos hacer es filtrar las tupias en las que el valor de A es simplemente una elipse incluso antes de intentar verificar el radio. Y esto es exactamente lo que sucede con la siguiente formulación:

```
R : IS_CIRCULO. ( A ) WHERE THER ( A ) > LONGITUD ( 2.0 )
```

Esta expresión está definida (en general) para que regrese aquellas tupias en las que el valor de A es un círculo con radio mayor que dos. Para ser más precisos, regresa una relación con

a. El mismo encabezado que R, con excepción de que el tipo declarado para el atributo A de ese resultado, es CIRCULO en lugar de ELIPSE;
b. Un cuerpo que consiste solamente en las tupias de R, en las que el valor de A es de tipo CIRCULO y el radio del círculo en cuestión es mayor que dos.

En otras palabras, de lo que estamos hablando es de un nuevo operador relacional de la forma

```
R : IS_T ( A )
```

donde *R* es una expresión relacional y A es un atributo de la relación (digamos *r)* indicado por esa expresión. El tipo declarado *DT{A)* de A debe ser un supertipo de *T* (ésta es una verificación en tiempo de compilación). El valor de la expresión general está definido para que sea una relación con:

a. Un encabezado igual al de *r;* salvo que el tipo declarado del atributo A en ese encabezado es *T;*
b. Un cuerpo que consiste en las tupias de *r* en las que el atributo *A* contiene un valor de tipo *T,* con excepción de que el tipo declarado para el atributo *A* en cada una de esas tupias es *T.*

En forma similar, también definimos otro nuevo operador de la forma

```
R : IS_MS_T ( A )
```

19.7 OPERADORES, VERSIONES Y SIGNATURAS

Recuerde de la sección 19.3 que un operador dado puede tener muchas *versiones de implementation* (también conocidas como **especializaciones explícitas)** bajo la mesa. Es decir conforme recorremos hacia abajo la ruta desde algún supertipo *T* hacia algún subtipo *T* en la jerarquía de tipos, necesitamos que al menos (por muchas razones) se nos *permita* reimplementar operadores de tipo 7* para el tipo *T'*. Como ejemplo considere el siguiente operador MOVER:

```
OPERATOR MOVER ( E ELIPSE, R RECTÁNGULO ) RETURNS ( ELIPSE )
   VERSION ERMOVER ;
   RETURN ( ELIPSE ( THE_A ( E ), THE_B ( E ), RCTRO ( R ) ) ) ;
END OPERATOR ;
```

En términos generales, el operador MOVER "mueve" la elipse E, para que esté ubicada en el centro del rectángulo R, o más precisamente, regresa una elipse similar a la elipse del argumento que corresponde con el parámetro E (salvo que está ubicada en el centro del rectángulo del argumento que corresponde con el parámetro R). Observe la cláusula VERSION de la segunda línea que presenta un nombre distintivo, ER_MOVER, para esta versión específica de MOVER (vea el siguiente párrafo). Observe también que hemos supuesto la disponibilidad de un operador R_CTRO que regresa el punto central de un rectángulo especificado.

Ahora supongamos que ha sido definida una especialización explícita —es decir otra versión— de MOVER para que mueva círculos en lugar de elipses:*

```
OPERATOR MOVER ( C CIRCULO, R RECTÁNGULO ) RETURNS ( CIRCULO )
   VERSION CR_MOVER ;
   RETURN ( CIRCULO ( THE_R ( C ) , R_CTRO ( R ) ) ) ;
END OPERATOR ;
```

En forma similar, tal vez también tengamos una especialización explícita para el caso en que los argumentos sean de tipos más específicos ELIPSE y CUADRADO, respectivamente (digamos EC_MOVER), y otra para el caso en donde los argumentos sean de los tipos más específicos CIRCULO y CUADRADO, respectivamente (digamos CC_MOVER).

Signaturas

En general, el término **signatura** significa la combinación del *nombre* de algún operador y los *tipos de operandos* del operador en cuestión. (Sin embargo, observamos de paso que los distintos autores y lenguajes le asignan significados ligeramente diferentes al término. Por ejemplo, el tipo del *resultado* es visto en ocasiones como parte de la signatura, así como en ocasiones lo son también los *nombres* de los operandos y del resultado.) Sin embargo, debemos ser muy cuidadosos nuevamente con:

a. la diferencia entre argumentos y parámetros;
b. la diferencia entre tipos declarados y tipos reales (más específicos); y

---

*De hecho, no tiene mucho caso definir tal especialización explícita en este ejemplo en particular (¿por qué exactamente?).

c. La diferencia entre los operadores tal como los ve el usuario y los operadores tal como los ve el sistema (lo que significa en el último caso, que las especializaciones o versiones explícitas de esos operadores existen bajo la mesa, como lo describimos anteriormente).

De hecho, podemos distinguir—¡aunque la literatura a menudo no lo hace!— al menos tres diferentes tipos de signaturas que están asociadas con cualquier operador dado *Op:*

- Una **signatura única de especificación,** que consiste en el nombre del operador *Op* junto con los tipos declarados (en orden) de los parámetros para *Op,* tal como el definidor de *Op* los especifica para el usuario. Esta signatura corresponde al operador *Op* tal como el usuario lo entiende. Por ejemplo, la signatura de especificación para MOVER es simplemente MOVER (ELIPSE, RECTÁNGULO).*
- Un conjunto de **signaturas de versión,** una para cada versión de especialización o implementación explícita de *Op,* donde cada una consiste en el nombre de operador *Op* junto con los tipos declarados (en orden) de los parámetros definidos para esa versión. Estas signaturas corresponden a las diferentes piezas de código de implementación que implementan a *Op* bajo la mesa. Por ejemplo, la signatura de versión para la versión CRJVIOVER de MOVER es: MOVER (CIRCULO, RECTÁNGULO).
- Un conjunto de **signaturas de invocación,** una para cada combinación posible de los tipos de argumento más específicos, donde cada una consiste en el nombre del operador *Op* junto con la combinación correspondiente de los tipos de argumento más específicos (en orden). Estas signaturas corresponden a las diversas invocaciones posibles de *Op* (por supuesto, la correspondencia es de uno a muchos; es decir una signatura de invocación puede corresponder a muchas invocaciones distintas). Por ejemplo, tengan E y R los tipos más específicos CIRCULO y CUADRADO, respectivamente. Entonces la signatura de invocación para la invocación MOVER(E, R) es: MOVER (CIRCULO, CUADRADO).

  De este modo, las distintas signaturas de invocación que involucran al mismo operador, corresponden (al menos potencialmente) a diferentes versiones implementadas del operador en cuestión; es decir, a especializaciones diferentes bajo la mesa. Por lo tanto, si en realidad existen varias versiones del "mismo" operador bajo la mesa, entonces la versión invocada en cualquier ocasión dada, dependerá de qué signatura de versión es "la que coincide mejor" para la signatura de invocación aplicable. El proceso de decidir esa coincidencia mejor—es decir, el proceso de decidir qué versión va a ser invocada— es por supuesto, el proceso de *enlace en tiempo de ejecución,* que ya explicamos en la sección 19.3.

---

*La referencia [3.3] propone que es posible separar la definición de la signatura de especificación para un operador dado de las definiciones de *todas* las implementaciones (versiones) de ese operador. La idea básica es soportar tipos *dummy* (también conocidos como tipos "abstractos" o "sin ejemplares" o, a veces, sólo como "interfaces"); es decir, tipos que no son el tipo más específico de valor alguno. Dichos tipos proporcionan una forma de especificar operadores que se aplican a varios tipos normales diferentes, en donde todos ellos son subtipos propios del tipo ficticio en cuestión. Entonces, un operador de éstos puede ser especializado —es decir, puede definir explícitamente una versión apropiada del operador— para cada uno de esos subtipos normales. En términos de nuestro ejemplo actual, FIGURA_PLANA bien podría ser un tipo ficticio en este sentido; la signatura de especificación del operador AREA bien podría ser definida en el nivel de FIGURA_PLANA, y el código de implementación explícito (versiones) ser definido para los tipos ELIPSE, POLÍGONO, etcétera.

Observe, de paso que (a) las signaturas de *especificación* son en realidad un concepto del modelo; (b) las signaturas de *versión* son simplemente un concepto de la implementación; (c) las signaturas de *invocación*, aunque en cierta forma son un concepto del modelo, son en realidad —al igual que el concepto de sustituibilidad— sólo una consecuencia lógica directa de la idea básica de herencia de tipo, en primer lugar. Además, el hecho de que sean posibles diferentes signaturas de invocación es en realidad sólo parte del concepto de sustituibilidad.

Operadores de sólo lectura y operadores de actualización

Hasta ahora hemos estado suponiendo tácitamente que MOVER es un operador de sólo lectura. Pero supongamos que fuéramos a hacer que en su lugar, fuera un operador de actualización:

```
OPERATOR MOVER ( E ELIPSE, R RECTÁNGULO ) UPDATES ( E )
   VERSION ER_MOVER ; BEGIN ;
THE_CTRO ( E ) : = R_CTRO ( R ) ; RETURN
; END ; END OPERATOR ;
```

(Le recordamos que a los operadores de sólo lectura y de actualización en ocasiones se les llama *observadores* y *matadores*, respectivamente. Consulte el capítulo 5 si necesita refrescar su memoria con respecto a la diferencia entre ellos.)

Observe ahora que una invocación a esta versión de MOVER *actualiza su primer argumento* (en general, "cambia el centro" de ese argumento). Observe además que la actualización funciona sin tomar en cuenta si el primer argumento es del tipo más específico ELIPSE o del tipo más específico CIRCULO; en otras palabras, no es necesaria la especialización explícita para los círculos.* Por lo tanto, una ventaja de los operadores de actualización (en general) es que pueden evitarnos la necesidad de escribir explícitamente determinadas especializaciones de operador. Observe en particular las implicaciones para el mantenimiento de programas; por ejemplo, ¿qué sucede si introducimos subsecuentemente a CIRCULO_O como subtipo de CIRCULO?

Cambio de la semántica del operador

El hecho de que al menos siempre sea válido volver a implementar los operadores conforme avanzamos hacia abajo en la jerarquía de tipos, tiene una consecuencia muy importante: abre la posibilidad de *cambiar la semántica* del operador en cuestión. Por ejemplo, en el caso de AREA, podría darse el caso de que la implementación para el tipo CIRCULO regresara en realidad (por decir algo) la circunferencia del círculo en cuestión, en lugar del área. (Un diseño cuidadoso del tipo puede ayudar a solucionar en parte este problema; por ejemplo, si el operador AREA está definido para que regrese un resultado de tipo AREA, la implementación obviamente no puede regresar un resultado de tipo LONGITUD en su lugar. Sin embargo, ¡todavía puede regresar el área *errónea*\)

---

*De hecho, es probable que ni sean necesarios si la especialización por restricción es soportada.

Además (aunque parezca sorprendente), podemos incluso afirmar —y de hecho, *ha* sido afirmado— que podría ser necesario cambiar de esta forma la semántica. Por ejemplo, sea el tipo AUTOPISTAJDE_CUOTA un subtipo propio del tipo AUTOPISTA, y sea TIEMPO_DE_V1AJE un operador que calcula el tiempo que lleva viajar entre dos puntos específicos de una autopista específica. Para una autopista de cuota la fórmula es *(d/v)* + (n*f); donde *d* = distancia, v = velocidad, *n* = número de casetas de cobro y í = tiempo transcurrido en cada caseta de cobro. Por el contrario, para una autopista que no es de cuota, la fórmula es simplemente *d/v.*

Como contraejemplo —es decir un ejemplo de una situación donde un cambio semántico es claramente innecesario— considere nuevamente las elipses y los círculos. Presuntamente, nos gustaría que el operador AREA estuviera definido en forma tal que un círculo dado tuviera la misma área sin tomar en cuenta si lo consideramos específicamente como círculo o sólo como una elipse. En otras palabras, supongamos que los siguientes eventos suceden en secuencia de la siguiente forma:

1. Definimos el tipo ELIPSE y una versión correspondiente del operador AREA. Suponemos, por razones de simplicidad, que el código de AREA no utiliza la representación *real* para las elipses.
2. Definimos el tipo CIRCULO como subtipo de ELIPSE, pero (todavía) no definimos una versión de implementación distinta de AREA para los círculos.
3. Invocamos a AREA sobre algún círculo específico *c* para obtener un resultado, digamos *áreal*. Por supuesto, esta invocación utiliza la versión de AREA para ELIPSE.
4. Ahora definimos una versión de implementación distinta de AREA para los círculos.
5. Invocamos nuevamente a AREA sobre el mismo círculo específico *c* (igual que antes) para obtener un resultado, digamos *áreal* (y esta vez es la versión de AREA de CIRCULO la que es llamada).

Luego, seguramente nos gustaría insistir en que *áreal = área2.* Sin embargo, no hay manera de hacer cumplir este requerimiento; es decir, como ya observamos, siempre existe la posibilidad de que la versión de AREA implementada para los círculos pueda regresar (digamos) la circunferencia en lugar del área, o simplemente el área errónea.

Regresemos al ejemplo de TIEMPO_DE_VIAJE. El hecho, en este ejemplo y otros similares, es extremadamente poco convincente; es decir no convincente como ejemplo de una situación en la cual el cambio de la semántica de un operador pudiera ser necesario. Considere lo siguiente:

- Si en realidad AUTOPISTA_DE_CUOTA es un subtipo de AUTOPISTA, significa por definición que toda autopista de cuota individual es de hecho una autopista.
- Por lo tanto, algunas autopistas (es decir, algunos valores de tipo AUTOPISTA) son de he cho autopistas de cuota; tienen casetas de cobro. Por lo tanto, AUTOPISTA no es "autopis tas sin casetas de cobro", sino que es "autopistas con *n* casetas de cobro" (en donde *n* puede ser cero).
- Por lo tanto, el operador TIEMPO_DE_VIAJE para el tipo AUTOPISTA no es "calcular el tiempo de viaje para una autopista *sin* casetas de cobro", sino que es "calcular el tiempo de viaje *d/v* para una autopista *ignorando* las casetas de cobro".

- El operador TIEMPO_DE_VIAJE para el tipo AUTOPISTA_DE_CUOTA es, por el contrario, "calcular el tiempo de viaje *(dlv) + (n*t)* para una autopista sin ignorar las casetas de cobro". Por lo tanto, los dos TIEMPO_DE_VIAJE son, de hecho, operadores diferentes desde un punto de vista lógico. Surge la confusión debido a que los dos operadores diferentes tienen el mismo nombre; de hecho, lo que estamos haciendo aquí es polimorfismo de sobrecarga y no polimorfismo de inclusión.

(Además, observamos que en la práctica surgen confusiones adicionales debido a que, por desgracia, muchos escritores en realidad usan el término *sobrecarga* para referirse también al polimorfismo de inclusión.)

Para resumir: todavía no creemos que cambiar la semántica del operador sea una buena idea. Como hemos visto, no es posible hacer cumplir este requerimiento, pero en realidad podemos definir nuestro modelo de herencia —y lo hacemos— para decir que si la semántica *es cambiada* entonces **la implementación está cometiendo una violación** (es decir, no es una implementación del modelo y las implicaciones son impredecibles). Observe que nuestra posición en este tema (es decir, nuestra posición de que tales cambios no son válidos) tiene la ventaja de que, sin tomar en cuenta si se definen algunas especializaciones explícitas de un operador *Op* dado, la percepción del usuario sigue siendo la misma. En concreto, (a) que existe un operador, un operador *único,* llamado *Op,* y (b) que el operador *Op* se aplica a los valores de argumento de algún tipo *T* especificado y por lo tanto —por definición— a valores de argumento de cualquier subtipo propio de *T.*

19.8 ¿UN CÍRCULO ES UNA ELIPSE?

¿En realidad los círculos son elipses? Hemos estado dando por hecho a lo largo de este capítulo —¡con mucha razón!— que sí lo son, pero ahora debemos enfrentar el hecho de que hay mucha controversia en la literatura sobre este punto aparentemente obvio. Considere nuestras variables usuales E y C de tipos declarados ELIPSE y CIRCULO, respectivamente. Supongamos que estas variables han sido iniciadas de la siguiente forma:

```
E := ELIPSE  ( LONGITUD ( 5.0 ), LONGITUD ( 3.0 ),
                                 PUNTO ( 0.0, 0.0 ) ) ; C
:= CIRCULO ( LONGITUD ( 5.0 ), PUNTO ( 0.0, 0.0 ) ) ;
```

En particular, observe que tanto THE_A(C) como THE_B(C) tienen el valor cinco.

Ahora, una operación que podemos realizar con seguridad sobre E es "actualizar el semieje *a";* por ejemplo:

```
THEA ( E ) := LONGITUD ( 6.0 ) ;
```

Pero si tratamos de realizar la operación equivalente en C

```
THEA ( C ) := LONGITUD ( 6.0 ) ;
```

¡obtenemos un error! ¿Qué tipo de error exactamente? Bien, si la actualización sucediera realmente, la variable C acabaría conteniendo un "círculo" que viola la restricción sobre círculos de que *a - b (a* sería ahora seis, mientras que *b* en apariencia seguiría siendo cinco, ya que no lo hemos cambiado). En otras palabras, C contendría ahora un "círculo no circular" lo cual viola por lo tanto la restricción de tipo sobre el tipo CIRCULO.

Debido a que los "círculos no circulares" son un insulto a la lógica y al sentido común, parece razonable sugerir que en primer lugar, no debemos permitir la actualización. Y la forma obvia para lograr este efecto es rechazar tales operaciones en tiempo de compilación, definiendo la actualización —es decir la asignación— de los semiejes *a* o *b* de un círculo para que sea *sintácticamente* inválido. En otras palabras, la asignación THE_A o THE_B no se aplica al tipo CIRCULO, y la actualización pretendida falla por un error de tipo *en tiempo de compilación*.

Nota: De hecho, es "obvio" que tales asignaciones deben ser sintácticamente inválidas. Recuerde que la asignación a una seudovariable THE_ en realidad sólo es una abreviatura. Entonces, por ejemplo, si la asignación intentada para THE_A(C) que mostramos anteriormente fuera inválida, tendría que ser una abreviatura de algo como esto:

```
C := CIRCULO ( . . . ) ;
```

Y la invocación al selector de CIRCULO del lado derecho tendría que incluir un argumento THE_A de LONGITUD(ó.O). Pero el selector de CIRCULO ¡no toma un argumento THE_A!, sino que toma el argumento THE_R y el argumento THE_CTRO. Por lo tanto, es claro que la asignación original debe ser inválida.

¿Qué hay sobre el cambio en la semántica?

Evitemos inmediatamente una sugerencia que en ocasiones se hace como un intento para rescatar la idea de que la asignación para THE_A o THE_B debe a fin de cuentas ser válida para los círculos. La sugerencia es que la asignación para (por ejemplo) THE_A debe ser *redefinida* —en otras palabras, *especializada explícitamente*— para un círculo, en forma tal que tenga también el efecto lateral de la asignación para THE_B, de manera que el círculo siga satisfaciendo la restricción *a = b* después de la actualización. Rechazamos esta sugerencia por (al menos) las siguientes tres razones:

- Primero, la semántica de la asignación para THE_A y THE_B es —¡deliberadamente!— prescrita por nuestro modelo de herencia y *no* debe ser cambiada de la manera sugerida.
- Segundo, aunque esa semántica no fuera prescrita por el modelo, ya hemos argumentado que (a) cambiar arbitrariamente la semántica de un operador es en general mala idea, y (b) cambiar la semántica de un operador, en forma tal que cause efectos laterales, es todavía peor. Es un buen principio general insistir que los operadores tienen exactamente el efecto solicitado, ni más ni menos.
- Tercero (y más importante), de cualquier forma, la opción de cambiar la semántica de la manera sugerida no está siempre disponible. Por ejemplo, hagamos que el tipo ELIPSE tenga otro subtipo inmediato NOCIRCULO; hagamos que la restricción *a > b* se aplique a los no círculos y considere una asignación THE_A para un no círculo que (de ser aceptada) podría hacer a *a* igual que *b*. ¿Cuál sería una redefinición semántica adecuada para esa asig nación? Exactamente, ¿qué efecto lateral sería adecuado?

¿Ha existido alguna vez un modelo razonable?

Por lo tanto, tenemos la situación en la que la asignación para THE_A o THE_B es una operación que se aplica a las elipses en general, pero no a los círculos en particular. Sin embargo:

a. se supone que el tipo CIRCULO es un subtipo del tipo ELIPSE;

b. al decir que el tipo CIRCULO es un subtipo del tipo ELIPSE queremos dar a entender que las operaciones que se aplican a las elipses en general, se aplican a —en otras palabras, son *heredadas* por— los círculos en particular;

c. pero ahora estamos diciendo que a fin de cuentas, la operación de la asignación para THE_A o THE_B *no* se hereda.

¿Tenemos entonces una contradicción en las manos? ¿Qué está pasando?

Antes de intentar contestar estas preguntas debemos enfatizar la seriedad del problema. El argumento anterior se parece a una auténtica madeja de hilos revueltos. Si determinados operadores *no* son heredados por el tipo CIRCULO a partir del tipo ELIPSE, ¿en qué sentido podemos decir exactamente que un círculo "es" una elipse? Después de todo, ¿qué significa "herencia" si algunos operadores de hecho no se heredan? ¿Ha existido alguna vez un modelo razonable de herencia? ¿Estamos buscando una quimera al tratar de encontrar uno?

Nota: Algunos autores han incluso sugerido —*seriamente*— que la asignación para THE_A debe funcionar tanto para círculos como para elipses (para un círculo, ésta actualiza el radio); mientras que la asignación para THE_B sólo debería funcionar para las elipses, y de esta forma ¡la ELIPSE en realidad debería ser un subtipo de CIRCULO! En otras palabras, tenemos de cabeza a la jerarquía de tipos. Sin embargo, un momento de razonamiento basta para mostrar que esta idea no tiene sentido; en particular, se rompería la sustituibilidad (¿cuál es el radio de una elipse general?).

Consideraciones como las anteriores, son las que han llevado a algunos autores a la conclusión de que en realidad, no existe un modelo de herencia razonable (vea el comentario a la referencia [19.1] en la sección "Referencias y bibliografía" al final del capítulo). Otros autores han propuesto modelos de herencia con características que son contraintuitivas o claramente indeseables. Por ejemplo, SQL3 permite "círculos no circulares" y otras cosas sin sentido; de hecho, al igual que SQL/92, no soporta en absoluto las restricciones de tipo, y es esta omisión la que permite que se presenten, en primer lugar, tales sinsentidos (vea el apéndice B).

La solución

Para resumir la situación que hemos visto hasta el momento, nos enfrentamos con el siguiente dilema:

- Si los círculos heredan los operadores "asignación para THE_A y THE_B" de las elipses, entonces obtenemos círculos no circulares.
- La forma de impedir círculos no circulares es soportar las restricciones de tipo.
- Pero si soportamos las restricciones de tipo, entonces no es posible heredar los operadores.
- Por lo tanto, ¡a fin de cuentas no hay herencia!

¿Cómo resolvemos este dilema?

La solución es, como sucede a menudo, reconocer (y actuar en consecuencia con) el hecho de que hay una gran diferencia lógica entre **valores y variables.** Cuando decimos "todo círculo es una elipse", lo que queremos decir es (más precisamente) que todo *valor* de círculo es un *valor* de elipse. En realidad no queremos decir que toda *variable* de círculo sea una *variable* de elipse (una variable de tipo declarado CIRCULO *no* es una variable de tipo declarado ELIPSE, y no

puede contener un valor del tipo más específico ELIPSE). En otras palabras, **la herencia** se **aplica a los valores y no a las variables.** En el caso de las elipses y los círculos, por ejemplo:

- Como acabamos de decir, todo valor de círculo es un valor de elipse.
- Por lo tanto, todas las operaciones aplicables a los valores de elipse, también son aplicables a los valores de círculo.
- Pero algo que no podemos hacerle a ningún valor es **¡cambiarlo!,** ya que de ser así ya no sería ese valor. (Por supuesto, podemos "cambiar el valor actual de" una variable actualizando esa variable, pero —para repetir— no podemos cambiar el valor como tal.)

Ahora bien, todas las operaciones que se aplican a los valores de elipse son precisamente operadores de *sólo lectura* para el tipo ELIPSE, mientras que las operaciones que actualizan las variables ELIPSE son por supuesto, los operadores de *actualización* definidos para ese tipo. Por lo tanto, nuestra aseveración de que "la herencia se aplica a los valores y no a las variables" puede ser expresada más precisamente de la siguiente manera:

- **Los operadores de sólo lectura son heredados por los valores y por lo tanto, *obligatoriamente* por los valores actuales de las variables** (ya que por supuesto, los operadores de sólo lectura pueden ser aplicados —sin peligro— a aquellos valores que representen los valores actuales de las variables).

Esta expresión más precisa también sirve para explicar por qué los conceptos de polimorfismo y sustituibilidad se refieren muy específicamente a los *valores* y no a las variables. Por ejemplo (y sólo para recordarlo), las sustituibilidad dice que cada vez que el sistema espera un **valor** de tipo *T* siempre puede sustituirlo por un **valor** de tipo 7", donde 7" es un subtipo de *T* (las negritas se añaden para enfatizarlo). De hecho, nos referíamos específicamente a este principio cuando lo presentamos por primera vez como el *principio de la sustituibilidad de valor* (nuevamente observe el énfasis).

Entonces, ¿qué hay acerca de los operadores de actualización? Por definición, dichos operadores se aplican a las variables y no a los valores. Por lo tanto, ¿podemos decir que los operadores de actualización que se aplican a las variables de tipo ELIPSE son heredados por las variables de tipo CIRCULO?

Bueno, no; no podemos (al menos no completamente). Por ejemplo, la asignación para THE_CTRO se aplica a las variables de ambos tipos declarados, pero (como hemos visto) la asignación para THE_A no se aplica. Por lo tanto, la herencia de los operadores de actualización tiene que ser *condicional;* de hecho, es necesario especificar explícitamente cuáles operadores de actualización se heredan. Por ejemplo:

- las variables de tipo declarado ELIPSE tienen operadores de actualización MOVER (en la versión de actualización) y de asignación para THE_A, THEJ5 y THE_CTRO;
- las variables de tipo declarado CIRCULO tienen operadores de actualización MOVER (versión de actualización) y de asignación para THE_CTRO y THE_R, pero *no* para THE_AoTHE_B.

Nota: Explicamos el operador MOVER en la sección anterior.

Por supuesto, si un operador de actualización *es* heredado, tenemos que hacer un tipo de polimorfismo y un tipo de sustituibilidad que sea aplicable a las variables y no solamente a los valores. Por ejemplo, la versión de actualización de MOVER espera un argumento que es una variable de tipo declarado ELIPSE, pero podemos invocarla con un argumento que sea una variable de tipo declarado CIRCULO. Por lo tanto, podemos (y lo hacemos) hablar sensatamente acerca

de un *principio de sustituibilidad de variable*, pero ese principio es más restrictivo que el *principio de sustituibilidad de valor* que tratamos anteriormente.

19.9 REVISIÓN DE LA ESPECIALIZACIÓN POR RESTRICCIÓN

Hay una pequeña pero significativa posdata que debemos añadir a la explicación de las secciones anteriores. Tiene que ver con ejemplos como este: "sea que el tipo CIRCULO tenga un subtipo propio llamado CIRCULO_COLOREADO" (lo que quiere decir que suponemos que "los círculos coloreados" son un caso especial de los círculos en general). Ejemplos de esta naturaleza son citados comúnmente en la literatura. Pero tenemos que decir que encontramos esos ejemplos extremadamente poco convincentes; incluso confusos, en ciertos aspectos importantes. Para ser más específicos, sugerimos en este caso que en realidad no tiene sentido pensar que los círculos coloreados son de alguna manera un caso especial de los círculos en general. Después de todo, (por definición) "los círculos coloreados" deben ser *imágenes* —tal vez en una pantalla— mientras que los círculos en general no son imágenes sino *figuras geométricas*. Por lo tanto, parece más razonable considerar a CIRCULO_COLOREADO no como un subtipo de CIRCULO sino en su lugar, como *un tipo completamente independiente*. Este tipo independiente podría tener una **representación posible** en la cual un componente sea de tipo CIRCULO y otro de tipo COLOR, pero no es (para repetir) un subtipo del tipo CIRCULO.

Herencia de representaciones posibles

El siguiente es un firme argumento que apoya la posición anterior. Primero, regresemos por un momento a nuestro ejemplo más usual de las elipses y los círculos. Éstas son de nuevo las definiciones de tipo (en parte):

```
TYPE ELIPSE POSSREP ( A LONGITUD, B LONGITUD, CTRO PUNTO ) ... ;

TYPE CIRCULO POSSREP ( R LONGITUD, CTRO PUNTO ) ... ;
```

En particular, observe que las elipses y los círculos tienen diferentes representaciones posibles declaradas. Sin embargo, la representación posible para las elipses es también —*necesaria* y además implícitamente— una representación posible para los círculos, debido a que los círculos *son* elipses. Es decir los círculos en realidad pueden estar "posiblemente representados" por sus semiejes *a y b* (y su centro), aunque de hecho, sus semiejes *a y b* sean iguales. Por supuesto, lo contrario no es cierto; es decir una representación posible para los círculos *no* es necesariamente una representación posible para las elipses.

Deducimos que es posible considerar a las representaciones posibles, al igual que a los operadores y las restricciones, como "propiedades" adicionales que son heredadas por los círculos desde las elipses o, más generalmente, por los subtipos desde los supertipos.* Pero (invirtiendo

---

*En nuestro modelo formal no las consideramos así —es decir, no consideramos tales representaciones posibles heredadas como *declaradas*— ya que si dijéramos que son declaradas nos llevaría a una contradicción. Para ser más específicos, si dijéramos que el tipo CIRCULO hereda una representación posible del tipo ELIPSE, entonces la referencia [3.3] requeriría la asignación THE_A o THE_B para que una variable de tipo declarado CIRCULO fuera válida; y por supuesto, sabemos que no es así. Por lo tanto, decir que el tipo CIRCULO hereda una representación posible del tipo ELIPSE es sólo *una manera de decirlo y* no tiene ningún peso formal.

ahora el caso de los círculos y los círculos coloreados) debe quedar claro que la representación posible declarada para el tipo CIRCULO *no* es una representación posible para el tipo CIRCULO_COLOREADO, ¡debido a que no hay nada en él que sea capaz de representar el color! Este hecho sugiere firmemente que los círculos coloreados *no* son círculos en el mismo sentido que, por ejemplo, los círculos y las elipses.

Entonces, ¿en realidad qué significan los subtipos?

El siguiente argumento se relaciona (en cierta forma) con el anterior; aunque es de hecho más fuerte (más fuerte *lógicamente,* claro). Este argumento es: *por medio de la especialización por restricción, no hay forma de obtener un círculo coloreado a partir de un círculo.*

Para explicar este punto debemos regresar por un momento al caso de las elipses y los círculos. Aquí tenemos una vez más las definiciones de tipo:

```
TYPE ELIPSE POSSREP ( A LONGITUD, B LONGITUD, CTRO PUNTO ) ...
     CONSTRAINT ( THEA ( ELIPSE ) > THEB ( ELIPSE ) ) ;

TYPE CIRCULO POSSREP ( R LONGITUD, CTRO PUNTO ) SUBTYPEOF (
     ELIPSE ) CONSTRAINT ( THE_A ( CIRCULO ) • THEB (
     CIRCULO ) ) ;
```

Como vimos anteriormente, la cláusula CONSTRAINT para el tipo CIRCULO garantiza que una elipse con $a = b$ será especializada automáticamente hacia el tipo CIRCULO. Pero —intercambiando ahora a los círculos y los círculos coloreados— no hay ninguna cláusula CONSTRAINT que podamos escribir para el tipo CIRCULO_COLOREADO que de igual forma haga que un círculo se especialice hacia el tipo CIRCULO_COLOREADO; es decir, no hay ninguna restricción de tipo que podamos escribir y que, cuando sea satisfecha por algún círculo dado, signifique que el círculo en cuestión en realidad sea un círculo coloreado.

Por lo tanto, parece de nuevo más razonable considerar a CIRCULO_COLOREAD0 y CIRCULO como tipos completamente diferentes, y en particular considerar al tipo C1RCULO_COLOREADO como que tiene una representación posible en la que un componente es de tipo CIRCULO y el otro es de tipo COLOR; de tal forma que:

```
TYPE CIRCUL0_C0L0READ0 POSSREP ( CIR CIRCULO, COL COLOR ) ... ;
```

De hecho, aquí estamos tocando un asunto mucho más grande. El hecho es que ¡creemos que el manejo de subtipos *siempre* debe ser por medio de la especialización por restricción! Es decir, sugerimos que **si *T'* es un subtipo de *T*, *siempre* debe haber una restricción de tipo tal que, si es satisfecha por algún valor dado del tipo *T*, entonces el valor en cuestión en realidad es un valor de tipo *T'*** (y debe ser especializado automáticamente al tipo 7"). Sean *TyT* tipos, y sea 7" un subtipo de *T* (de hecho, podemos suponer, sin perder la generalidad, que *T* **es** : un subtipo *inmediato* de 7). Entonces:

- *T*y *T* son básicamente *conjuntos* (conjuntos de valores nombrados) y *V* es un subconjunto *deT*.
- Por lo tanto, *TyT'* tienen *predicados de pertenencia;* es decir predicados tales que un valor sea un miembro del conjunto en cuestión (y por lo tanto, un valor del tipo en cuestión) si y sólo si satisface el predicado en cuestión. Sean esos predicados *P* y *P',* respectivamente.
- Observe ahora que el predicado *P'* es, por definición, un predicado que puede dar como resul tado *verdadero* sólo para determinados valores que en realidad son valores de tipo *T.* Enton ces, puede ser formulado en términos de valores de tipo *T* (en lugar de valores de tipo *T).*

- Y ese predicado *P',* formulado en términos de los valores del tipo T, es precisamente la restricción de tipo que tienen que satisfacer los valores de tipo Tpara que sean valores de tipo 7". En otras palabras, un valor de tipo *T* es especializado hacia el tipo 7" precisamente si satisface la restricción *P'.*

Por lo tanto, podemos afirmar que la especialización por restricción es el *único* medio conceptualmente válido para definir subtipos. Por consecuencia, rechazamos ejemplos como el que sugiere que CIRCULO_COLOREADO puede ser un subtipo de CIRCULO.

9.10 RESUMEN

Hemos bosquejado los conceptos básicos del **modelo de herencia de tipo.** Si el tipo *B* es un subtipo del tipo A (y en forma equivalente, el tipo A es un supertipo del tipo *B),* entonces todo valor de tipo *B* también es un valor de tipo *A y* por lo tanto, lo operadores y restricciones que se aplican a los valores del tipo A también se aplican a los valores del tipo *B* (aunque también habrá operadores y restricciones que se aplican a valores del tipo *B* pero que no se aplican a valores que sean solamente del tipo A). Distinguimos la herencia **simple** contra la **múltiple** (pero sólo tratamos la herencia simple) y la herencia **escalar** contra la de **tupia** y contra la de **relación** (pero sólo tratamos la herencia escalar) para presentar el concepto de **jerarquía de tipos.** También definimos los términos subtipo y supertipo **propios,** subtipo y supertipo **inmediatos,** tipo **raíz** y tipo **hoja,** y establecimos una **suposición de tipos disjuntos:** los tipos *TI y T2* son disjuntos a menos que uno sea un subtipo del otro. Como consecuencia de esta suposición, todo valor tiene un tipo único **más específico** (sin ser necesariamente un tipo hoja).

Después explicamos los conceptos de polimorfismo (de inclusión) y **sustituibilidad** (de valor), los cuales son consecuencias lógicas de la noción básica de herencia. Distinguimos entre polimorfismo de **inclusión** (que tiene que ver con la herencia) y polimorfismo de **sobrecarga** (que no tiene que ver con ella). Y mostramos la manera en que, gracias al **enlace en tiempo de ejecución,** el polimorfismo de inclusión puede conducir a la **reutilización del código.**

Luego consideramos los efectos de la herencia sobre las operaciones de **asignación.** El punto básico es que las conversiones de tipo *no* suceden —los valores conservan su tipo más específico en la asignación hacia variables de tipo declarado menos específico— y por lo tanto, una variable de tipo declarado *T* puede tener un valor cuyo tipo más específico sea cualquier subtipo de *T.* (En forma similar, si el operador *Op* está definido para tener un resultado de tipo declarado *T,* el resultado actual de una invocación de *Op* puede ser un valor cuyo tipo más específico sea cualquier subtipo del tipo *T.)* Por lo tanto, modelamos una variable escalar *V*—o más generalmente, una expresión escalar arbitraria— como un triple ordenado de la forma <*DT, MST,* v>, donde *DT* es el tipo declarado, *MST* es el tipo más específico actual y v es el valor actual. Presentamos el operador **TREAT DOWN** para que nos permitiera operar —en formas tales que de no ser así, provocarían un error de tipo en tiempo de compilación— sobre expresiones cuyo tipo más específico en tiempo de ejecución es algún subtipo propio de su tipo declarado. (Es posible que ocurran errores de tipo en tiempo de ejecución, pero sólo dentro del contexto de TREAT DOWN.)

Después vimos de cerca a los **selectores.** Vimos que en algunas ocasiones, la invocación de un selector para el tipo *T* producirá un resultado de algún subtipo propio de *T* (al menos en

nuestro modelo, aunque por lo general, no en los productos comerciales actuales); es decir, la **especialización por restricción.** Luego dimos un vistazo cercano a las **seudovariables THE_;** debido a que en realidad sólo son abreviaturas, en una asignación a una seudovariable THE_ puede ocurrir tanto la especialización como la **generalización** por restricción.

Luego continuamos explicando los efectos de los subtipos y supertipos en las **comparaciones de igualdad** y en determinadas operaciones relaciónales **(junta, unión, intersección y diferencia).** También presentamos varios operadores para **prueba de tipo: IS_7\ IS_MS_r,** etcétera. También consideramos la cuestión de los operadores de **sólo lectura** contra los de **actualización,** las **versiones** de un operador y las **signaturas** de un operador, y resaltamos que la habilidad para definir diferentes versiones de un operador abre las puertas para el **cambio** de **la semántica** del operador en cuestión (aunque nuestro modelo prohibe dichos cambios).

Por último, analizamos la pregunta "¿son los círculos en realidad elipses?". Ese análisis nos condujo a la posición de que **la herencia se aplica a los valores** y **no a las variables.** Para ser más precisos, los operadores de sólo lectura (que se aplican a los valores) pueden ser heredados al cien por ciento sin ningún problema, pero los operadores de actualización (que se aplican a las variables) pueden ser heredados sólo **condicionalmente.** (Nuestro modelo está en desacuerdo con la mayoría de los demás enfoques. Por lo general, esos otros enfoques requieren que los operadores sean heredados incondicionalmente, pero luego sufren por una variedad de problemas que tienen que ver con los "círculos no circulares" y cosas similares.) Y concluimos comentando que en nuestra opinión, la especialización por restricción es la *única* forma lógicamente válida para la definición de subtipos.

EJERCICIOS

19.1 Defina los siguientes

enlace en tiempo de ejecución	subtipo propio
especialización por restricción	sustituibilidad
generalización por restricción	tipo escalar
polimorfismo	tipo ficticio (dummy)
reutilización del código	tipo hoja
signatura	tipo no escalar
subtipo inmediato	tipo raíz

19.2 Explique el operador TREAT DOWN.

19.3 Establezca las diferencias entre:

a. argumento y parámetro
b. tipo declarado y tipo más específico actual
c. polimorfismo de inclusión y polimorfismo de sobrecarga
d. signatura de invocación, signatura de especificación y signatura de versión
e. operador de sólo lectura y operador de actualización
f. valor y variable

19.4 Con respecto a la jerarquía de tipos de la figura 19.1, considere un valor *e* de tipo ELIPSE. El tipo más específico de *e* es ELIPSE o CIRCULO. ¿Cuál es el tipo *menos* específico de *el*

19.5 Cualquier jerarquía de tipos dada, incluye varias subjerarquías que pueden ser consideradas como jerarquías de tipos por derecho propio. Por ejemplo, la jerarquía obtenida borrando (solamente) los tipos FIGURA_PLANA, ELIPSE y CIRCULO —a partir de la jerarquía de la figura 19.1— puede ser considerada como una jerarquía de tipos por derecho propio; y lo mismo sucede para la jerarquía obtenida al borrar (solamente) los tipos CIRCULO, CUADRADO y RECTÁNGULO. Por otro lado, la jerarquía obtenida borrando (solamente) a ELIPSE, *no puede* ser considerada como jerarquía de tipos por derecho propio (al menos, ninguna que pueda derivar de la figura 19.1), ya que el tipo CIRCU LO "pierde algo de su herencia", en esa jerarquía. Entonces, ¿cuántas jerarquías de tipos distintas están juntas en la figura 19.1?

19.6 Utilice la sintaxis esbozada en el cuerpo del capítulo, para dar las definiciones de tipo para los tipos RECTÁNGULO y CUADRADO (para simplificar, suponga que todos los rectángulos están centrados en el origen, pero sin suponer que todos los lados son verticales u horizontales).

19.7 Dada su respuesta al ejercicio 19.6, defina un operador para rotar un rectángulo especificado 90° alrededor de su centro. También dé una especialización explícita de ese operador para los cuadrados.

19.8 Ésta es una repetición del ejemplo de la sección 19.6: "La varrel R tiene un atributo A de tipo declarado ELIPSE, y queremos consultar a R para obtener las tupias donde el valor de A sea en rea lidad un círculo y el radio de ese círculo sea mayor que dos." Dimos la siguiente formulación a esta consulta en la sección 19.6:

```
R : ISCIRCULO ( A ) WHERE THE_R ( A ) > LONGITUD ( 2.0 )
```

a. ¿Por qué no podemos simplemente expresar la prueba de tipo en la cláusula WHERE?, por ejemplo:

```
R WHERE IS_CIRCULO ( A ) AND THE_R ( A ) > LONGITUD ( 2.0 )
```

b. Otra formulación candidata es:

```
R WHERE CASE
           WHEN IS_CIRCULO ( A ) THEN
                THE_R ( TREAT_DOWN_AS_CIRCULO ( A ) )
                                    > LONGITUD ( 2.0 )
           WHEN NOT ( ISCIRCULO ( A ) ) THEN FALSE
        END CASE
```

¿Es válida? Si no lo es, ¿por qué?.

19.9 La referencia [3.3] propone el soporte para expresiones relaciónales de la forma

```
fl TREAT_DOWN_AS  7" ( A )
```

Aquí *R* es una expresión relacional, *A* es un atributo de la relación (digamos *r)* indicada por esa expresión y *T* es un tipo. El tipo declarado *DT(A)* de *A* debe ser un supertipo de *T* (ésta es una verificación en tiempo de compilación). El valor de la expresión general está definido para que sea una relación con:

a. Un encabezado igual al de r, excepto que el tipo declarado del atributo *A* en ese encabezado es *T*.
b. Un cuerpo que contiene las mismas tupias que *r,* salvo que el valor de *A* en cada una de esas tu pias ha sido degradado al tipo *T*.

Sin embargo, este operador es (de nuevo) simplemente una abreviatura, ¿de qué exactamente?

19.10 Las expresiones de la forma *R:IS_T(A)* también son abreviaturas, ¿de qué exactamente?

REFERENCIAS Y BIBLIOGRAFÍA

19.1 Malcolm Atkinson *et al:* "The Object-Oriented Database System Manifesto", Proc. First Inter national Conference on Deductive and Object-Oriented Databases, Kyoto, Japón (1989). Nueva York, N.Y.: Elsevier Science (1990).

Con relación a la falta de consenso (como mencionamos en la sección 19.1) sobre un buen modelo de herencia, los autores del presente texto dicen esto: "[hay] al menos cuatro tipos de herencia: de *sustitución,* de *inclusión,* de *restricción* y de *especializarían...* Varios grados de estos cuatro tipos de herencia son proporcionados por los sistemas y prototipos existentes, y no prescribimos un estilo específico de herencia".

Éstas son algunas citas más que ilustran el mismo punto general:

- Cleaveland [19.4] dice: "[la herencia puede estar] basada en [una variedad de] criterios di ferentes y no hay una definición estándar aceptada comúnmente"; y continúa dando ocho interpretaciones posibles. (Meyer [19.8] proporciona doce.)
- Baclawski e Indurkhya [19.2] dicen: "[un] lenguaje de programación proporciona [simple mente] un conjunto de mecanismos [de herencia]. Aunque estos mecanismos ciertamente restringen lo que uno puede hacer en ese lenguaje y qué vistas de la herencia pueden ser implementadas... no validan, por sí mismos, una u otra vista de la herencia. Las clases, las especializaciones, las generalizaciones y la herencia son solamente conceptos, y... no tienen un significado objetivo universal... Este [hecho] implica que la manera en que se incorpora la herencia en un sistema específico queda en manos de los diseñadores de [ese] sistema y cons tituye una decisión política que debe ser implementada con los mecanismos disponibles". En otras palabras, ¡no hay modelo!

19.2 Kenneth Baclawski y Bipin Indurkhya: Technical Correspondence, *CACM 37,* No. 9 (septiem bre, 1994).

19.3 Luca Cardelli y Peter Wegner: "On Understanding Types, Data Abstraction, and Polymor phism", *ACM Comp. Surv. 17,* No. 4 (diciembre, 1985).

19.4 J. Craig Cleaveland: *An Introduction to Data Types.* Reading, Mass.: Addison-Wesley (1986).

19.5 C. J. Date: Serie de artículos sobre la herencia de tipo en el sitio web de *DBP&D www.dbpd.com* (primera instalación en febrero de 1999).

Es un amplio tratamiento tutorial (con notas históricas) del modelo de herencia que describimos en el presente capítulo y está definido más formalmente en la referencia [3.3].

19.6 C. J. Date y Hugh Darwen: "Toward a Model of Type Inheritance", *CACM 41,* No. 12 (di ciembre, 1998).

Esta breve nota incluye un resumen de las características principales de nuestro modelo de herencia.

19.7 Nelson Mattos y Linda G. DeMichiel: "Recent Design Trade-Offs in SQL3", *ACM SIGMOD Record 23,* No. 4 (diciembre, 1994).

Este artículo da las razones de las decisiones que tomaron los diseñadores de SQL3 para no soportar las restricciones de tipo (está basado en los argumentos dados primeramente por Zdonik y Maier en la referencia [19.11]). Sin embargo, no estamos de acuerdo con esas razones. El problema fundamental contra ellas es que fallan para distinguir adecuadamente entre valores y variables (vea el ejercicio 19.3).

19.8 Bertrand Meyer: " The Many Faces of Inheritance: A Taxonomy of Taxonomy", *IEEE Com puter 29,* No. 5 (mayo, 1996).

19.9 James Rumbaugh: "A Matter of Intent: How to Define Subclasses", *Journal of Object-Oriented Programming* (septiembre, 1996).

Como dijimos en la sección 19.9, nuestro punto de vista es que la especialización por restricción es la única manera lógicamente válida para definir subtipos. Por lo tanto, ¡es interesante observar que el mundo de los objetos toma exactamente la posición opuesta! (o al menos, algunos habitantes de ese mundo lo hacen). Para citar a Rumbaugh: "¿Es CUADRADO una subclase de RECTÁNGULO?... El alargamiento de la dimensión *x* de un rectángulo es algo perfectamente razonable. Pero si se hace lo mismo con un cuadrado, entonces el objeto ya no es un cuadrado. Esto no es algo malo conceptualmente. Cuando se alarga un cuadrado *se obtiene* un rectángulo... Pero... la mayoría de los lenguajes orientados a objetos no quieren que los objetos cambien de clase... Todo esto sugiere [un] principio de diseño para los sistemas de clasificación: *una subclase no debe ser definida mediante la restricción de una superclase"* (cursivas en el original). *Nota:* Como explicamos en el capítulo 24, el mundo de los objetos usa frecuentemente el término *clase* para referirse a lo que nosotros llamamos *tipo.*

Nos sorprende que Rumbaugh tome aparentemente esta posición simplemente porque los lenguajes orientados a objetos "no quieren que los objetos cambien de clase". Preferiríamos tener primero el *modelo* antes de preocuparnos acerca de las implementaciones.

19.10 Andrew Taivalsaari: "On the Notion of Inheritance", *ACM Comp. Surv. 28,* No. 3 (septiembre, 1996).

19.11 Stanley B. Zdonik y David Maier: "Fundamentals of Object-Oriented Databases", en la referencia [24.52].

STAS A EJERCICIOS SELECCIONADOS

19.3 Consideramos solamente el caso f. (valor y variable), ya que es fundamental y todavía no lo hemos tratado explícitamente en alguna otra parte del libro. (Las siguientes definiciones están tomadas de la referencia [3.3].)

- Un **valor** es una "constante individual" (por ejemplo, la constante individual "3"). Un valor *no tiene ubicación en el tiempo ni en el espacio.* Sin embargo, los valores pueden estar representa dos en la memoria mediante alguna *codificación* y por supuesto, tales codificaciones sí tienen ubicaciones en el tiempo y en el espacio (vea el siguiente párrafo). Observe que, por definición, *un valor no puede ser actualizado;* si pudiera serlo, después de tal actualización ya no sería ese valor (en general).
- Una **variable** es un receptáculo para la codificación de un valor. Una variable tiene una ubicación en el tiempo y en el espacio. Por supuesto, también las variables, a diferencia de los valores, pueden ser actualizadas; es decir, el valor actual de la variable en cuestión puede ser reemplazado por otro valor, probablemente diferente al anterior. (Por supuesto, la variable en cuestión es la misma después de la actualización.)

Dicho sea de paso, es importante comprender que no sólo cosas simples como el entero "3" son valores legítimos. Por el contrario, los valores pueden ser arbitrariamente complejos; por ejemplo, un valor puede ser un arreglo, o una pila, o una lista, o una relación, o un punto geométrico, o una elipse, o un rayo X, o un documento, o una huella digital (etcétera). Por supuesto, también se aplican consideraciones similares a las variables.

19.4 Por supuesto, el tipo menos específico de cualquier valor de *cualquiera* de los tipos que mues tra la figura 19.1 es FIGURA_PLANA.

19.5 22 (esta cuenta incluye la jerarquía vacía).

19.6 Puesto que todos los rectángulos están centrados en el origen, un rectángulo dado ABCD puede ser identificado en forma única por medio de dos vértices adyacentes (digamos A y B) y un cuadrado

dado puede ser identificado en forma única por cualquier vértice (digamos A). Para especificarlo más precisamente, tomamos a A como el vértice que está en el cuadrante superior derecho del plano *(x* S: 0, y > 0) y a B como el vértice que está en el cuadrante inferior derecho (í^Oјí 0). Luego podemos definir a los tipos RECTÁNGULO y CUADRADO de la siguiente forma:

```
TYPE RECTÁNGULO POSSREP ( A PUNTO, B PUNTO ) . . . ;

TYPE CUADRADO POSSREP ( A PUNTO )
     CONSTRAINT ( THE_X ( THE_A ( CUADRADO ) ) =
                - THE_Y ( THE_B ( CUADRADO ) ) AND
                  THE_Y ( THE_A ( CUADRADO ) ) ■
                  THE_X ( THE_B ( CUADRADO ) ) ) ;
```

19.7 Los operadores que definimos a continuación son específicamente operadores de actualización. Como ejercicio adicional, defina algunos de sólo lectura similares a éstos.

```
OPERATOR ¡OTAR ( R RECTÁNGULO   UPDATES ( R
 VERSION ROTAR RECTÁNGULO ¡
  BEGIN ;
     VAR P PUNTO ;
     VAR Q PUNTO ;
     P : = = THE A ( R ) ;
     ° :▪▪ = THE B ( R ) ;
     THE X ( THE A ( R ) ) :=      - Y ( Q
     THF" "Y ( THE A ( R ) ) : =  THE X ( Q
     THE X ( THE B ( R ) ) :=     THE_ " ( P
     THF  ~Y ( THE B ( R ) ) : =     - X ( P
     RETURN ;
  END ;
END OPERATOR :
```

```
OPERATOR ROTAR ( C CUADRADO ) UPDATES ( C )
   VERSION R0TAR_CUADRAD0 ;
   RETURN ;
END OPERATOR ;
```

19.8

a. La expresión especificada fallará por un error de tipo en tiempo de compilación, debido a que THE_R requiere un argumento de tipo CIRCULO y el tipo declarado de A es ELIPSE, no CIRCULO. (Por supuesto, si no se realiza la verificación en tiempo de compilación, en vez de ello obtendremos un error de tipo en *tiempo de ejecución* tan pronto como se encuentre una tupia en la que el valor de A sea simplemente una elipse y no un círculo.)

b. La expresión especificada es válida, pero produce una relación con el mismo encabezado que R, y no una en el que el tipo declarado del atributo A sea CIRCULO en lugar de ELIPSE.

19.9 La expresión es una abreviatura para una expresión de la forma

```
( ( EXTEND ( fl ) ADD ( TREAT_DOWN_AS T" ( A ) ) AS A' )
                                  { ALL BUT A } ) RENAME A' AS A
```

(donde A' es un nombre cualquiera que no aparece como nombre de atributo en el resultado de la evaluación de *R).*

19.10 La expresión es una abreviatura para una expresión de la forma

```
( R WHERE IS_7 ( A ) ) TREAT_OOWN_AS_r ( A )
```

Además, esta última expresión es en sí misma, una abreviatura para otra más larga, tal como vimos en la respuesta del ejercicio 19.9.

CAPITULO **20**

Bases de datos distribuidas

20.1 INTRODUCCIÓN

Tocamos el tema de las **bases de datos distribuidas** al final del capítulo 2, donde dijimos que (para citar) "el soporte completo para las bases de datos distribuidas implica que una sola aplicación debe ser capaz de operar de manera transparente sobre los datos que están dispersos en una variedad de bases de datos diferentes, administradas por una variedad de distintos DBMSs, ejecutadas en diversas máquinas diferentes, manejadas por varios sistemas operativos diferentes y conectadas a una variedad de redes de comunicación distintas; donde el término *de manera transparente* significa que la aplicación opera desde un punto de vista lógico como si todos los datos fueran manejados por un solo DBMS y ejecutados en una sola máquina". Ahora estamos en posición de analizar estas ideas con mayor detalle. Para ser específicos, en este capítulo explicaremos exactamente lo que es una base de datos distribuida, la razón por la que tales bases de datos están llegando a ser cada vez más importantes, así como algunos de los problemas técnicos en el campo de las bases de datos distribuidas.

El capítulo 2 también trató brevemente a los sistemas **cliente-servidor,** los cuales pueden ser considerados como un caso especial sencillo de los sistemas distribuidos en general. En la sección 20.5 consideraremos específicamente a los sistemas cliente-servidor.

Explicamos el plan general del capítulo al final de la siguiente sección.

20.2 ALGUNOS PUNTOS PRELIMINARES

Comenzamos con una definición funcional (un poco imprecisa en este momento):

- Un sistema de base de datos distribuida consiste en una colección de **sitios,** conectados por medio de algún tipo de red de comunicación, en el cual
  a. cada sitio es un sistema de base de datos completo por derecho propio, pero
  b. los sitios han acordado trabajar juntos, a fin de que un usuario de cualquier sitio pueda acceder a los datos desde cualquier lugar de la red, exactamente como si los datos estu vieran guardados en el propio sitio del usuario.

De aquí deducimos que la llamada "base de datos distribuida" es en realidad un tipo de base de datos *virtual* cuyas partes componentes están almacenadas en varias bases de datos "reales" distintas que se encuentran en varios sitios distintos (de hecho, es la unión lógica de esas bases de datos reales). La figura 20.1 muestra un ejemplo.

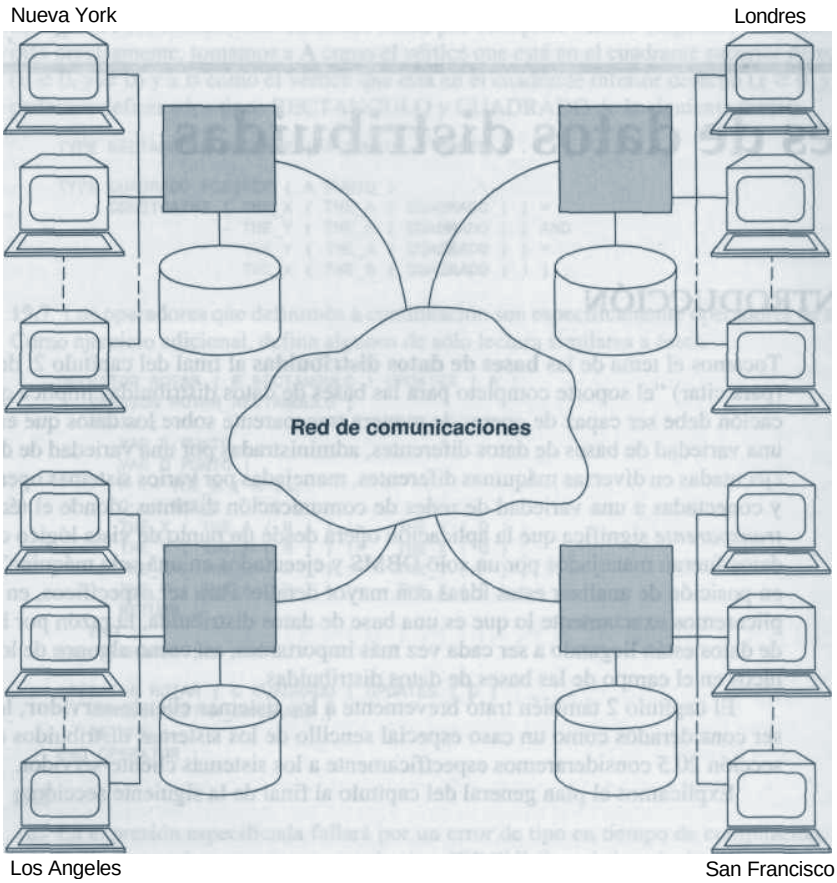


Figura **20.1** Un sistema de base de datos distribuida típico.

Para repetir, observe que **cada sitio es un sitio de sistema de base de datos por derecho propio.** En otras palabras, cada sitio tiene sus propias bases de datos "reales", sus propios usuarios locales, su propio DBMS local y software de administración de transacciones (incluyendo su propio software local para bloqueo, registro en bitácora, recuperación, etcétera), así como su propio administrador CD (de comunicación de datos) local. En particular, un usuario determinado puede realizar operaciones sobre los datos desde su propio sitio local, tal como si ese sitio no participara nunca en el sistema distribuido (al menos, éste es un objetivo). Entonces, el sistema de base de datos distribuida puede ser considerado como un tipo de **sociedad** entre los DBMSs locales en cada uno de los sitios locales; un nuevo componente de software en cada sitio —de manera lógica, una extensión del DBMS local— proporciona la funcionalidad de sociedad necesaria, y es la combinación de este nuevo componente y el DBMS existente, lo que constituye lo que generalmente llamamos **sistema de administración de base de datos distribuida.**

Dicho sea de paso, es común suponer que los sitios componentes están dispersos físicamente —quizá también dispersos geográficamente, como lo sugiere la figura 20.1—, aunque de hecho basta con que estén dispersos *lógicamente.* Dos "sitios" pueden incluso coexistir en la misma máquina física (especialmente durante el período de la prueba inicial del sistema). De hecho, el énfasis sobre los sistemas distribuidos ha ido y venido a lo largo del tiempo. Las primeras investigaciones tendían a dar por hecho la distribución geográfica; sin embargo la mayoría de las primeras instalaciones comerciales involucraban la distribución *local* con (por ejemplo) varios "sitios" en el mismo edificio y conectados por medio de una LAN (red de área local). Sin embargo, más recientemente la enorme proliferación de las WANs (redes de área amplia) ha revivido nuevamente el interés en la posibilidad de una distribución geográfica. En cualquier caso, desde una perspectiva de base de datos, hay muy poca diferencia —ya que es necesario resolver prácticamente los mismos problemas técnicos (de la base de datos)— y por lo tanto, para efectos de este capítulo podemos ver razonablemente a la figura 20.1 como una representación de un sistema típico.

Nota: Para simplificar la explicación supondremos, mientras no digamos otra cosa, que el sistema es *homogéneo* en el sentido de que cada sitio está ejecutando una copia del mismo DBMS. Nos referiremos a esto como la suposición de **homogeneidad estricta.** En la sección 20.6 exploraremos la posibilidad —y algunas de las implicaciones— de la flexibilización de esta suposición.

Ventajas

¿Por qué son necesarias las bases de datos distribuidas? La respuesta básica a esta pregunta es que las empresas ya están generalmente distribuidas al menos de manera lógica (en divisiones, departamentos, grupos de trabajo, etcétera) y es muy probable que también lo estén de manera física (en plantas, fábricas, laboratorios, etcétera); de esto deducimos que por lo general también los datos ya están distribuidos, ya que cada unidad organizacional dentro de la empresa mantendrá naturalmente los datos que son importantes para su propia operación. Por lo tanto, el valor de la información total de la empresa está dividido en lo que a veces llamamos ***islas de información.*** Y lo que hace un sistema distribuido es proporcionar los *puentes* necesarios para conectar a esas islas entre sí. En otras palabras, permite que la estructura de la base de datos refleje la estructura de la empresa —los datos locales son conservados localmente en el lugar donde pertenecen de manera más lógica— y al mismo tiempo, permite tener acceso a datos remotos cuando sea necesario.

Un ejemplo le ayudará a aclarar lo anterior. Considere de nuevo la figura 20.1. Por razones de simplicidad, suponga que hay solamente dos sitios, Los Angeles y San Francisco, y suponga que es un sistema bancario donde los datos de las cuentas de Los Angeles son conservados en Los Angeles y los datos de las cuentas de San Francisco son conservados en San Francisco. Las ventajas son claramente obvias: el arreglo distribuido combina la **eficiencia de procesamiento** (los datos se mantienen cerca del punto en donde se usan más frecuentemente) con una **mayor** accesibilidad (es posible acceder a una cuenta de Los Angeles desde San Francisco y *viceversa,* por medio de la red de comunicaciones).

Probablemente, el mayor beneficio de los sistemas distribuidos es que permiten que la estructura de la base de datos refleje la estructura de la empresa (como acabamos de explicar). Por supuesto, también se acumulan muchos beneficios adicionales, pero dejamos la explicación de estos beneficios adicionales para los puntos correspondientes del capítulo. Sin embargo, debemos

mencionar que también hay algunas desventajas; entre ellas, la mayor es el hecho de que los sistemas distribuidos son *complejos* (al menos desde un punto de vista técnico). Por supuesto, de manera ideal esa complejidad debe ser problema del implementador y no del usuario, pero es probable —siendo pragmáticos— que algunos aspectos aparecerán ante los usuarios, a menos que se tomen precauciones muy cuidadosas.

Sistemas de ejemplo

Para efectos de una referencia posterior, mencionaremos brevemente algunas de las implementaciones más conocidas de los sistemas distribuidos. Primero consideraremos a los prototipos. Entre muchos sistemas de investigación, tres de los más conocidos son (a) *SDD-1*, que fue construido en la división de investigación de Computer Corporation of America a finales de los aflos setenta y principios de los ochenta [20.34]; (b) *R** ("R estrella"), una versión distribuida del prototipo System R, construida en IBM Research a principios de los años ochenta [20.39]; y (c) *Distributed Ingres*, una versión distribuida del prototipo Ingres, construida también a principios de los ochenta en la Universidad de California en Berkeley [20.36].

En lo que respecta a las implementaciones comerciales, la mayoría de los productos SQL actuales proporcionan algún tipo de soporte de base de datos distribuida (con diversos grados de funcionalidad, por supuesto). Algunos de los más conocidos incluyen (a) *Ingres/Star*, el componente de base de datos distribuida de Ingres; (b) la *opción de base de datos distribuida* de Oracle; y (c) *Impropiedad de datos distribuidos* de DB2. *Nota:* Es obvio que estas dos listas de sistemas no pretenden ser completas, sino que simplemente pretenden identificar determinados sistemas que han sido o están siendo particularmente influyentes por una u otra razón, o que tienen algún interés especial intrínseco.

Vale la pena señalar que todos los sistemas que mencionamos anteriormente, tanto los prototipos como los productos, son relaciónales (al menos todos soportan SQL). Además, hay varias razones por las cuales, para que un sistema distribuido sea exitoso, *debe* ser relacional; la tecnología relacional es un requisito previo para la tecnología distribuida (efectiva) [20.15]. Conforme avancemos en el capítulo veremos algunas de las razones de esta situación.

Un principio fundamental

Ahora podemos establecer lo que puede ser considerado como **el principio fundamental de la base de datos distribuida** [20.14]:

- *Ante el usuario, un sistema distribuido debe lucir exactamente igual que un sistema que no es distribuido.*

En otras palabras, los usuarios de un sistema distribuido deben ser capaces de comportarse exactamente como si el sistema *no* fuera distribuido. Todos los problemas de los sistemas distribuidos son, o deberían ser, problemas internos o en el nivel de implementación, y no externos o en el nivel del usuario.

Nota: El término "usuarios" en el párrafo anterior se refiere específicamente a los usuarios (usuarios finales o programadores de aplicaciones) que están realizando operaciones de *manipulación de datos*. Todas las operaciones de manipulación de datos deben permanecer lógicamente sin cambios. Por el contrario, las operaciones de *definición* de datos requerirán de alguna

extensión en un sistema distribuido; por ejemplo, para que un usuario que esté en el sitio *X* pueda especificar que una varrel dada sea dividida en "fragmentos" que van a estar guardados en los sitios *Y* y Z (vea la explicación sobre la fragmentación en la siguiente sección).

El principio fundamental identificado anteriormente nos conduce a doce reglas complementarias u objetivos,* los cuales serán explicados en la siguiente sección. Aquí listamos esos objetivos como referencia:

1. Autonomía local.
2. No dependencia de un sitio central.
3. Operación continua.
4. Independencia de ubicación.
5. Independencia de fragmentación.
6. Independencia de replicación.
7. Procesamiento de consultas distribuidas.
8. Administración de transacciones distribuidas.
9. Independencia de hardware.
10. Independencia de sistema operativo.
11. Independencia de red.
12. Independencia de DBMS.

Tome en cuenta, que *no* todos estos objetivos son independientes entre sí; tampoco son necesariamente exhaustivos y tampoco son todos igual de importantes (diferentes usuarios darán diferentes grados de importancia a diferentes objetivos en diferentes ambientes; además, algunos de ellos pueden ser totalmente inaplicables en algunas situaciones). Sin embargo, los objetivos son útiles como una base para la comprensión de la tecnología distribuida y como un marco de trabajo para la caracterización de la funcionalidad de sistemas distribuidos específicos. Por lo tanto, los usaremos como un principio de organización para el resto del capítulo. La sección 20.3 presenta una breve exposición de cada objetivo; luego la sección 20.4 se enfoca en determinados asuntos específicos con mayor detalle. La sección 20.5 (como dijimos anteriormente) trata a los sistemas cliente-servidor. La sección 20.6 analiza a profundidad el objetivo específico de la independencia del DBMS. Por último, la sección 20.7 trata la cuestión del soporte dado por SQL, y la sección 20.8 proporciona un resumen y unas cuantas conclusiones.

Un punto final introductorio: es importante distinguir a los sistemas de *bases de datos* distribuidas verdaderos y generalizados, de los sistemas que simplemente proporcionan algún tipo de *acceso a datos remotos* (que es todo lo que en realidad hacen los sistemas cliente-servidor). En un sistema de acceso a datos remotos, el usuario puede operar simultáneamente sobre datos que están en un sitio remoto o incluso en varios sitios remotos, pero "se ven las costuras"; el usuario está definitivamente consciente, en mayor o menor grado, de que los datos son remotos

---

*"Reglas" fue el término usado en el artículo donde se presentaron por primera vez [20.14] (y el "principio fundamental" fue mencionado como *regla cero).* Sin embargo, el término "objetivos" en realidad resulta más adecuado; "reglas" suena demasiado dogmático. En el presente capítulo nos mantendremos con el término más suave "objetivos".

y que tiene que comportarse de acuerdo con ello. Por el contrario, en un verdadero sistema de base de datos distribuida las costuras están ocultas. (Gran parte de este capítulo se refiere a lo que significa, en este contexto, decir que las costuras están ocultas.) En el resto del capítulo usaremos el término "sistema distribuido" para referirnos específicamente a un sistema de base de datos distribuida verdadero y generalizado (en oposición a un simple sistema de acceso a datos remotos), a menos que indiquemos explícitamente lo contrario.

20.3 LOS DOCE OBJETIVOS

1. Autonomía local

Los sitios en un sistema distribuido deben ser **autónomos.** La autonomía local significa que todas las operaciones en un sitio dado están controladas por ese sitio; ningún sitio *X* debe depender de algún otro sitio F para su operación satisfactoria (ya que de lo contrario, cualquier falla en el sitio y podría significar que el sitio *X* no pueda ejecutar operaciones, aun cuando no haya nada malo en el propio sitio *X,* lo que es obviamente una situación indeseable). La autonomía local también implica que los datos locales son poseídos y administrados localmente con contabilidad local: todos los datos pertenecen "realmente" a alguna base de datos local, aun cuando estén accesibles desde otros sitios remotos. Por lo tanto, la seguridad, integridad y representación de almacenamiento de los datos locales permanecen bajo el control y jurisdicción del sitio local.

De hecho, el objetivo de autonomía local no es totalmente alcanzable; existen varias situaciones en las que un sitio *X* dado *debe* transferir un determinado grado de control a algún otro sitio y. Por lo tanto, el objetivo de autonomía queda establecido con mayor precisión como: los sitios deben ser autónomos **en el mayor grado posible.** Para más detalles vea el comentario de la referencia [20.14].

2. No dependencia de un sitio central

La autonomía local implica que **todos los sitios deben ser tratados como iguales.** Por lo tanto, no debe haber particularmente ninguna dependencia de un sitio "maestro" central para algún servicio central —por ejemplo, procesamiento centralizado de consultas, administración centralizada de transacciones o servicios centralizados de asignación de nombres— tal que todo el sistema dependa de ese sitio central. Por lo tanto, este segundo objetivo es un corolario del primero (si el primero se logra, el segundo también *forzosamente).* Pero "la no dependencia de un sitio central" es necesaria por sí misma, aunque no se logre la autonomía local completa. Por lo tanto, vale la pena mencionarlo como un objetivo independiente.

La dependencia de un sitio central sería indeseable por las siguientes dos razones (como mínimo). Primero, el sitio central puede ser un cuello de botella. Segundo y más importante, el sistema sería *vulnerable;* es decir, si el sitio central falla, también fallará todo el sistema (el problema del "único punto de falla").

3. Operación continua

En general, una ventaja de los sistemas distribuidos es que deben proporcionar mayor *confiabilidad* y mayor *disponibilidad.*

- La **confiabilidad** (es decir, la probabilidad de que el sistema esté listo y funcionando en cualquier momento dado) mejora, ya que los sistemas distribuidos no son una propuesta de todo o nada; pueden continuar operando (en un nivel reducido) cuando hay alguna falla en algún componente independiente, tal como un sitio individual.
- La **disponibilidad** (es decir, la probabilidad de que el sistema esté listo y funcionando con tinuamente a lo largo de un período especificado) también mejora, en parte por la misma razón y en parte debido a la posibilidad de replicación de datos (más adelante, vea el comen tario adicional en el número 6).

Lo anterior se aplica al caso en el que ocurre un **apagado no planeado** (es decir, una falla de algún tipo) en algún punto del sistema. Los apagados no planeados son obviamente indeseables, ¡pero es difícil evitarlos! Por el contrario, los apagados **planeados** *nunca* deberían ser requeridos; es decir, nunca debería ser necesario apagar el sistema para realizar alguna tarea como la incorporación de un nuevo sitio o la actualización del DBMS a una nueva versión en un sitio existente.

4. Independencia de ubicación

La idea básica de la **independencia de ubicación** (también conocida como **transparencia** de ubicación) es simple. Los usuarios no tienen que saber dónde están almacenados físicamente los datos, sino que deben ser capaces de comportarse —al menos desde un punto de vista lógico— como si todos los datos estuvieran almacenados en su propio sitio local. La independencia de ubicación es necesaria debido a que simplifica los programas de usuario y las actividades terminales. En particular, permite que los datos emigren de un sitio a otro sin invalidar ninguno de estos programas o actividades. Dicha capacidad de emigración es necesaria, ya que permite mover los datos por la red en respuesta a los diferentes requerimientos de desempeño.

Nota: Sin duda alguna, se dará cuenta de que la independencia de ubicación es simplemente una extensión al caso distribuido del concepto familiar de la independencia de *datos* (física). De hecho —para adelantarnos por un momento— cada objetivo de nuestra lista que tenga "independencia" en su nombre, puede ser considerado como una extensión a la independencia de datos, como veremos más adelante. Tendremos un poco más que decir con relación a la independencia de ubicación específicamente en la sección 20.4 (en nuestra explicación del *nombramiento de objetos* dentro de la subsección titulada "Administración del catálogo").

5. Independencia de fragmentación

Un sistema soporta la **fragmentación de datos** cuando una varrel dada puede ser dividida en partes *o fragmentos,* para efectos de almacenamiento físico. La fragmentación es necesaria por razones de rendimiento: los datos pueden estar almacenados en la ubicación donde son usados más frecuentemente para que la mayoría de las operaciones sean locales y se reduzca el tráfico en la red. Por ejemplo, considere una varrel de empleados EMP con los valores de ejemplo que muestra la parte superior de la figura 20.2. En un sistema que soporta la fragmentación, podríamos definir dos fragmentos de la siguiente forma:

```
FRAGMENT EMP AS
   N_EMP AT SITE 'Nueva York' WHERE DEPTO# = 'D1'
                               OR   DEPTO# = 'D3',
   L EMP AT SITE 'Londres'    WHERE DEPTO# - 'D2';
```

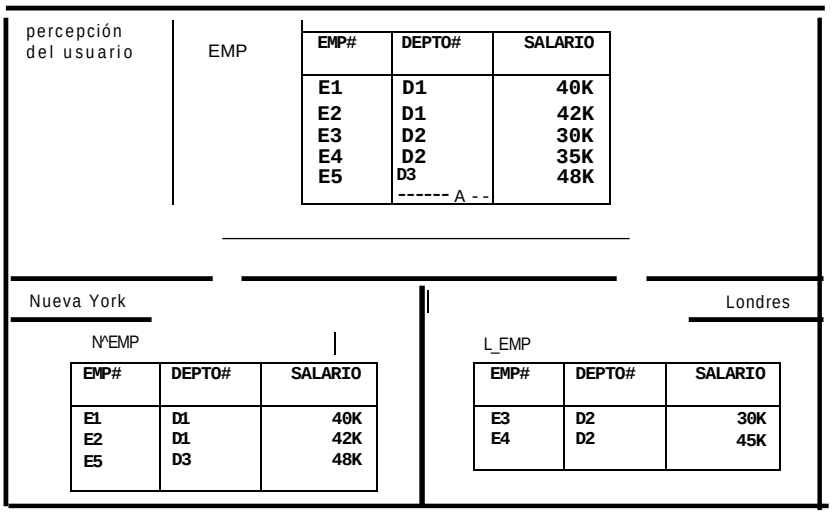


Figura 20.2 Un ejemplo de fragmentación.

(consulte la parte inferior de la figura 20.2). *Nota:* Estamos suponiendo que (a) las tupias de empleado se transforman hacia un almacenamiento físico en alguna forma bastante directa; (b) los números de empleado y los números de departamento son simplemente cadenas de caracteres, y los salarios son simplemente números; (c) DI y D3 son los departamentos de Nueva York y D2 es un departamento de Londres. Por lo tanto, las tupias para los empleados de Nueva York están almacenadas en el sitio de Nueva York y las tupias para los empleados de Londres están almacenadas en el sitio de Londres. Observe los nombres de fragmento internos del sistema N_EMP y L_EMP. Existen básicamente dos tipos de fragmentación, *horizontal* y *vertical,* que corresponden a las operaciones relaciónales de restricción y proyección, respectivamente (la figura 20.2 muestra una fragmentación horizontal). En términos más generales, un fragmento puede ser derivado mediante cualquier combinación arbitraria de restricciones y proyecciones; arbitraria con excepción de que:

- en el caso de la restricción, las restricciones deben constituir una descomposición *ortogonal,* en el sentido que mencionamos en el capítulo 12 (sección 12.6);
- en el caso de la proyección, las proyecciones deben constituir una descomposición *sin pér dida,* en el sentido que mencionamos en los capítulo 11 y 12.

El efecto neto de estas dos reglas es que todos los fragmentos de una varrel dada serán *independientes,* en el sentido de que ninguno de ellos puede ser derivado a partir de los otros ni tienen una restricción o proyección que puede ser derivada a partir de los otros. (Si en realidad queremos guardar la misma pieza de información en varios lugares diferentes, podemos usar el mecanismo de *replicación* del sistema; vea la siguiente subsección.)

La reconstrucción de la varrel original a partir de los fragmentos es lograda por medio de las operaciones de junta y de unión adecuadas (junta para los fragmentos verticales y unión para

los horizontales). Dicho sea de paso, en el caso de unión observe que no será necesario eliminar duplicados gracias a la primera de las dos reglas anteriores.

Nota: Debemos comentar brevemente el asunto de la fragmentación vertical. Como afirmamos antes, es cierto que dicha fragmentación debe ser sin pérdida, y por lo tanto, no será válida la fragmentación de la varrel EMP en sus proyecciones (digamos) {EMP#, DEPTO#} y {SALARIO}. Sin embargo, en algunos sistemas se considera que las varrels almacenadas tienen un "ID de tupia" o atributo **IDT** oculto, proporcionado por el sistema (donde el IDT para una tupia dada almacenada es en términos generales la *dirección* de la tupia en cuestión). Ese atributo IDT es claramente una clave candidata para la varrel aplicable; entonces, si (por ejemplo) la varrel EMP incluye este atributo, *podría* ser fragmentada válidamente en sus proyecciones {IDT, EMP#, DEPTO#} e {IDT, SALARIO}, debido a que esta fragmentación es claramente sin pérdida. Observe también que el hecho de que el atributo IDT esté oculto, no viola el *principio de información;* ya que la independencia de fragmentación significa que el usuario no está consciente de la fragmentación en forma alguna.

Además, observe que la facilidad de fragmentación y la facilidad de reconstrucción son dos de las razones principales por las que los sistemas distribuidos son relaciónales; el modelo relacional proporciona exactamente las operaciones necesarias para estas tareas [20.15].

Ahora llegamos al punto principal. Un sistema que soporta la fragmentación de datos también debe soportar la **independencia de fragmentación** (conocida también como **transparencia** de fragmentación); es decir, los usuarios deben ser capaces de comportarse, al menos desde un punto de vista lógico, como si los datos en realidad estuvieran sin fragmentación alguna. La independencia de fragmentación (al igual que la independencia de ubicación) es necesaria debido a que simplifica los programas de usuario y las actividades terminales. En particular, permite que los datos sean fragmentados en cualquier momento (y los fragmentos sean redistribuidos en cualquier momento) en respuesta a los diferentes requerimientos de rendimiento, sin invalidar ninguno de esos programas de usuario o actividades.

La independencia de fragmentación implica que a los usuarios se les presentará una vista de los datos en la cual los fragmentos estarán recombinados lógicamente por medio de juntas y de uniones adecuadas. Es responsabilidad del *optimizador del sistema* determinar cuáles fragmentos necesitan ser accedidos físicamente para satisfacer cualquier solicitud de usuario dada. Por ejemplo, dada la fragmentación que muestra la figura 20.2, si el usuario hace la solicitud

```
EMP WHERE SALARIO > 40K AND DEPT0# = ' D1 '
```

el optimizador sabrá, por medio de las definiciones de fragmentos (guardadas en el catálogo, por supuesto), que el resultado completo puede ser obtenido desde el sitio de Nueva York; no hay necesidad de acceder al sitio de Londres.

Examinemos este ejemplo un poco más a fondo. La varrel EMP, tal como es percibida por el usuario, puede ser considerada (aproximadamente) como una **vista** de los fragmentos subyacentes N_EMP y L_EMP:

```
VAR EMP VIEW
    N_EMP UNION L_EMP ;
```

El optimizador transforma entonces la solicitud original del usuario en lo siguiente:

```
( N_EMP UNION L_EMP ) WHERE SALARIO > 40K AND DEPTO# = 'D1
```

Esta expresión puede ser transformada todavía más en:

```
( N_EMP WHERE SALARIO > 40K AND DEPTO# = 'D1')
  UNION ( L_EMP WHERE SALARIO > 40K AND DEPT0# =
'D1')
```

(ya que la restricción se distribuye sobre la unión). A partir de la definición del fragmento L_EMP en el catálogo, el optimizador sabe que el segundo de estos dos operandos de UNION da como resultado una relación vacía (la condición de la restricción DEPTO# = 'DI' AND DEPTO# = 'D2' nunca puede dar como resultado *verdadero).* La expresión general puede entonces ser simplificada a sólo

```
N_EMP WHERE SALARIO > 40K AND DEPTO# = 'D1'
```

Ahora el optimizador sabe que necesita acceder solamente al sitio de Nueva York. *Ejercicio:* Considere lo que está involucrado en el optimizador para que maneje la solicitud

```
EMP WHERE SALARIO > 40K
```

Como lo sugiere la parte anterior, el problema de manejar operaciones sobre varrels fragmentadas tiene determinados puntos en común con el problema de manejar operaciones sobre vistas de junta y de unión (de hecho, los dos problemas son uno mismo; sólo que están manifestados en puntos diferentes de la arquitectura general del sistema). En particular, el problema de la *actualización* de varrels fragmentadas es el mismo que el problema de la actualización de las vistas de junta y de unión (vea el capítulo 9). Podemos deducir también que la actualización de una tupia dada (hablando de nuevo en términos generales) puede ocasionar que una tupia emigre de un fragmento hacia otro, en caso de que la tupia actualizada ya no satisfaga el predicado del fragmento al que pertenecía anteriormente.

6. Independencia de replicación

Un sistema soporta la **replicación de datos** cuando una varrel almacenada dada —o más generalmente, un *fragmento* dado de una varrel guardada— puede ser representada por muchas copias distintas, o *réplicas,* guardadas en muchos sitios distintos. Por ejemplo:

```
REPLICATE N_EMP AS
   LN_EMP AT SITE 'Londres' ;

REPLICATE L_EMP AS
   NL_EMP AT SITE 'Nueva York' ;
```

(vea la figura 20.3). Observe los nombres de réplica internos del sistema NL_EMP y LN_EMP.

Las réplicas son necesarias por dos razones (como mínimo). Primero, puede significar un mejor rendimiento (las aplicaciones pueden operar sobre copias locales en lugar de tener que comunicarse con sitios remotos). Segundo, también puede significar una mejor disponibilidad (un objeto replicado dado permanece disponible para su procesamiento —al menos para su recuperación— mientras esté disponible al menos una copia). Por supuesto, la principal desventaja de la replicación es que al actualizar un objeto replicado dado es necesario actualizar *todas las copias* de ese objeto: el problema de la **propagación de la actualización.** En la sección 20.4 tendremos más que decir con relación a este problema.

Comentamos de paso que la replicación en un sistema distribuido representa una aplicación específica de la idea de la *redundancia controlada,* tal como explicamos en el capítulo 1.

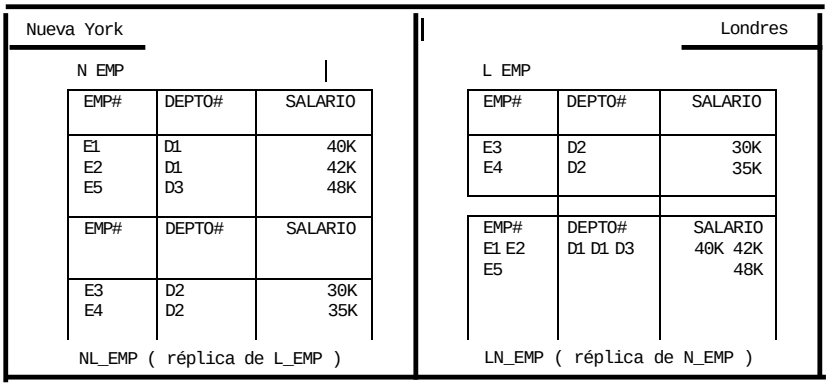


Figura 20.3 Un ejemplo de replicación.

Ahora bien (de manera ideal), la replicación, al igual que la fragmentación, debe ser "transparente ante el usuario". En otras palabras, un sistema que soporta la replicación de datos también debe soportar la **independencia de replicación** (también conocida como **transparencia** de replicación); es decir, los usuarios deben ser capaces de comportarse —al menos desde un punto de vista lógico— como si los datos en realidad no estuvieran replicados. La independencia de replicación (al igual que la independencia de ubicación y la de fragmentación) es necesaria debido a que simplifica los programas de usuario y las actividades terminales; en particular, permite que las réplicas sean creadas y destruidas en cualquier momento en respuesta a los distintos requerimientos, sin invalidar ninguno de esos programas de usuario o actividades.

La independencia de replicación implica que es responsabilidad del optimizador del sistema determinar cuáles réplicas necesitan ser accedidas físicamente para satisfacer cualquier solicitud de usuario dada. Aquí omitimos los puntos específicos de este tema.

Cerramos esta subsección mencionando que muchos productos comerciales soportan actualmente una forma de replicación que *no* incluye la independencia de replicación plena (es decir, *no* es completamente "transparente ante el usuario"). Vea los comentarios adicionales sobre este tema en la sección 20.4, dentro de la subsección sobre la propagación de la actualización.

7. Procesamiento de consultas distribuidas

Hay dos temas amplios a tratar bajo este encabezado.

- Primero, considere la consulta "obtener los proveedores de partes rojas de Londres". Suponga que el usuario está en el sitio de Nueva York y los datos están en el sitio de Londres. Suponga también que hay *n* proveedores que satisfacen la solicitud. Si el sistema es relacional, la consulta involucrará básicamente dos mensajes: uno para enviar la solicitud desde Nueva York a Londres y otro para regresar el conjunto resultante de *n* tupias desde Londres a Nueva York. Por otro lado, si el sistema no es relacional, sino que es un sistema de un registro a la vez, la consulta involucrará básicamente 2« mensajes: *n* desde Nueva York

hacia Londres solicitando al "siguiente" proveedor y *n* desde Londres hacia Nueva York para regresar ese "siguiente" proveedor. El ejemplo ilustra entonces el hecho de que es probable que un sistema relacional supere a uno que no lo es por varios órdenes de magnitud posibles.

Segundo, la **optimización** es todavía más importante en un sistema distribuido que en uno centralizado. El punto básico es que en una consulta como la anterior —que involucra a varios sitios— habrá muchas formas posibles de mover los datos en el sistema para satisfacer la solicitud, y es crucialmente importante que se encuentre una estrategia eficiente. Por ejemplo, una solicitud de (por decir algo) la unión de una relación *Rx* almacenada en el sitio *X* y una relación *Ry* almacenada en el sitio *Y,* podría ser realizada moviendo *Rx* hacia *Y* o *Ry* hacia *X,* o moviendo ambas hacia un tercer sitio Z (etcétera). En la sección 20.4 presentamos una ilustración convincente de este punto, que involucra a la consulta que mencionamos anteriormente ("obtener los números de proveedor de los proveedores de partes rojas de Londres "), Para resumir brevemente el ejemplo, analizamos seis estrategias diferentes para el procesamiento de la consulta bajo un conjunto determinado de suposiciones admisibles, y mostramos que el tiempo de respuesta ¡varía desde el mínimo de un décimo de segundo hasta el máximo de casi *seis horasl* Por lo tanto, la optimización es claramente crucial, y este hecho puede ser considerado a su vez como otra razón por la que los sistemas distribuidos siempre son relaciónales (donde el punto importante es que las peticiones relaciónales son optimizables, mientras que las no relaciónales no lo son).

8. Administración de transacciones distribuidas

Como usted sabe, hay dos aspectos principales en la administración de transacciones: el control de la recuperación y el control de la concurrencia. Ambos requieren un tratamiento amplio en el ambiente distribuido. Para explicar ese tratamiento amplio, primero es necesario introducir un nuevo término, el *agente.* En los sistemas distribuidos, una sola transacción puede involucrar la ejecución de código en muchos sitios; en particular, puede involucrar actualizaciones en muchos sitios. Por lo tanto, decimos que cada transacción consiste en varios agentes, donde un agente es el proceso realizado en nombre de una transacción dada en un sitio dado. Y el sistema necesita saber cuándo dos agentes son parte de la misma transacción; por ejemplo, no se debe permitir que dos agentes que son parte de la misma transacción caigan en bloqueo mortal entre ellos.

Continuemos específicamente con el control de la recuperación. Para asegurarse de que una transacción dada sea atómica (todo o nada) en el ambiente distribuido, el sistema debe por lo tanto asegurarse de que el conjunto de agentes para esa transacción sea confirmado o deshecho al unísono. Este efecto puede lograrse por medio del protocolo de **confírmación de dos** fases que ya explicamos (aunque no en el contexto distribuido) en el capítulo 14. En la sección 20.4 tendremos más que decir con relación a la confirmación de dos fases para un sistema distribuido.

En lo que se refiere al control de concurrencia, en la mayoría de los sistemas distribuidos —al igual que los no distribuidos— el control de concurrencia está basado generalmente en el **bloqueo.** (Varios productos recientes han comenzado a implementar *controles multiversión* [15.1]; sin embargo, en la práctica el bloqueo convencional parece seguir siendo la técnica a escoger para la mayoría de los sistemas.) De nuevo, en la sección 20.4 trataremos este tema con mayor detalle.

9. Independencia de hardware

De hecho, no hay mucho que decir sobre este tema, ya que el encabezado lo dice todo. Por lo general, las instalaciones de computadoras involucran en la realidad una gran diversidad de máquinas diferentes —IBM, ICL, HP, PC y estaciones de trabajo de diversos tipos, etcétera— y existe una necesidad real de poder integrar los datos en todos estos sistemas y presentar al usuario una "imagen de sistema único". Por lo tanto, es necesario tener la posibilidad de ejecutar el mismo DBMS en diferentes plataformas de hardware y, además, hacer que esas máquinas diferentes participen como socios igualitarios en un sistema distribuido.

10. Independencia de sistema operativo

En parte, este objetivo es un corolario del anterior y tampoco requiere de mucha explicación aquí. Obviamente es necesario no sólo tener la posibilidad de ejecutar el mismo DBMS en diferentes plataformas de hardware, sino también ejecutarlo en diferentes plataformas de sistema operativo —incluyendo diferentes sistemas operativos en el mismo hardware— y tener (por ejemplo) una versión MVS, una versión UNIX y una versión NT participando en el mismo sistema distribuido.

11. Independencia de red

De nuevo no hay mucho que decir sobre este punto; si el sistema va a tener la posibilidad de soportar muchos sitios distintos —con hardware distinto y sistemas operativos distintos— es obviamente necesario tener la posibilidad de soportar también una variedad de redes de comunicación distintas.

12. Independencia de DBMS

Bajo este encabezado consideramos lo que está involucrado al flexibilizar la suposición de homogeneidad estricta. Esta suposición es —discutiblemente— un poco más fuerte; todo lo que en realidad necesitamos es que **todos** los ejemplares del DBMS en sitios diferentes **soporten la misma interfaz,** aunque no tienen que ser necesariamente copias del mismo software DBMS. Por ejemplo, si Ingres y Oracle soportan el estándar de SQL oficial, sena posible tener un sitio Ingres y un sitio Oracle comunicándose entre sí en el contexto de un sistema distribuido. En otras palabras, sería posible que el sistema distribuido fuera *heterogéneo,* al menos en cierto grado.

El soporte para la heterogeneidad es definitivamente necesario. El hecho es que por lo general las instalaciones de computación en la realidad emplean no sólo muchas máquinas diferentes y muchos sistemas operativos diferentes, sino que muy frecuentemente, también DBMSs diferentes; y sería muy bueno si esos DBMS diferentes pudieran participar de alguna forma en un sistema distribuido. En otras palabras, el sistema distribuido ideal debe proporcionar **independencia de** DBMSs.

Sin embargo, éste es un tema grande (y uno muy importante en la práctica) al que le dedicamos una sección aparte. Vea posteriormente la sección 20.6.

20.4 PROBLEMAS DE LOS SISTEMAS DISTRIBUIDOS

En esta sección tratamos un poco más a fondo algunos de los problemas que mencionamos brevemente en la sección 20.3. El problema principal es que las redes de comunicación —al menos las de "larga distancia" o redes de área amplia (WANs) — son *lentas.* Una WAN típica puede tener una velocidad de datos efectiva cercana a los 5 o 10 mil bytes por segundo; por el contrario, la unidad de disco típica tiene una velocidad de datos cercana a los 5 o 10 *millones* de bytes por segundo. (Por otro lado, algunas redes de área local soportan velocidades de datos de la misma magnitud que las unidades de disco.) Por consecuencia, un objetivo principal en los sistemas distribuidos (al menos en el caso de las WAN y hasta cierto punto también en el caso de las LAN) es **minimizar la utilización de la red;** es decir, minimizar la cantidad y el volumen de los mensajes. Este objetivo hace a su vez que se presenten problemas en varias áreas complementarias, entre ellas las siguientes (esta lista no pretende ser completa):

- Procesamiento de consultas.
- Administración del catálogo.
- Propagación de la actualización.
- Control de la recuperación.
- Control de la concurrencia.

Procesamiento de consultas

El objetivo de minimizar la utilización de la red implica que el propio proceso de optimización de consultas debe ser distribuido, al igual que el proceso de ejecución de la consulta. En otras palabras, el proceso de optimización general consistirá típicamente en un paso de **optimización global** seguido por pasos de **optimización local** en cada sitio afectado. Por ejemplo, supongamos que una consulta *Q* es enviada al sitio *X,* y supongamos que *Q* involucra la unión de una relación *Ry* de cien tupias en el sitio *Y* con una relación *Rz* de un millón de tupias en el sitio Z. El optimizador que está en el sitio *X* seleccionará la estrategia global para la ejecución de *Q;* y es claramente necesario que decida mover *Ry* hacia Z y no *Rz* hacia *Y* (por supuesto, tampoco *Ry* y *Rz* hacia *X).* Entonces, una vez que ha decidido mover *Ry* hacia Z, la estrategia para ejecutar la unión real en el sitio Z será decidida por el optimizador local que está en Z.

La siguiente ilustración más detallada del punto anterior está basada en un ejemplo dado en la referencia [20.13], que fue adoptado a su vez de un artículo anterior de Rothnie y Goodman [20.33].

- *Base de datos* (proveedores y partes, simplificada):

```
v   { V#,  CIUDAD }                    10,000 tupias almacenadas en el sitio A
P   { P#,  COLOR }                    100,000 tupias almacenadas en el sitio B
VP { V#,   P# }                   1,000,000 de tupias almacenadas en el sitio A
```

  Suponga que cada tupia almacenada es de 25 bytes (200 bits) de longitud.

- *Consulta* ("obtener los números de proveedor de los proveedores de partes rojas de Londres"):

```
( ( V JOIN VP JOIN P ) WHERE CIUDAD = 'Londres' AND
                             COLOR = COLOR ( 'Rojo' ) ) { V# }
```

- *Cardinalidades estimadas de ciertos resultados intermedios:*

  Cantidad de partes rojas =10
  Cantidad de envíos de los proveedores de Londres = 100,000

- *Suposiciones de comunicación:*

  Velocidad de datos = 50,000 bits por segundo
  Retardo en el acceso = 0.1 segundos

Ahora analicemos brevemente seis estrategias posibles para el procesamiento de esta consulta, y para cada estrategia *i* calcularemos el tiempo de comunicación total *T[i]* a partir de la fórmula

(retardo de acceso total) + (volumen **total** de datos / velocidad de datos)

que se convierte (en segundos)

(cantidad de mensajes / 10) + (cantidad de bits / 50800)

1. Mover partes hacia el sitio A y procesar la consulta en A.

   7"[1] - 0.1 + (100000 * 200) / 50000
   = 400 segundos aproximadamente (6.67 minutos)

2. Mover proveedores y envíos hacia el sitio B y procesar la consulta en B.

   T[2] = 0.2 + **((10000** + **1000000)** • 200) / 50000
   = 4040 segundos aproximadamente (1.12 horas)

3. Juntar proveedores y envíos en el sitio A, restringir el resultado a los proveedores de Lon dres y luego, para cada uno de esos proveedores, revisar el sitio B para ver si la parte corres pondiente es roja. Cada una de estas revisiones involucrará dos mensajes, una consulta y una respuesta. El tiempo de transmisión para estos mensajes será pequeño en comparación con el retardo del acceso.

   7[3] = 20000 segundos aproximadamente (5.56 horas)

4. Restringir las partes en el sitio B a solamente las que sean rojas y luego, para cada una de estas partes, revisar el sitio A para ver si existe un envío que relacione la parte con un proveedor de Londres. Cada una de estas revisiones involucrará dos mensajes; de nuevo, el tiempo de transmisión de los mensajes será pequeño en comparación con el retardo del acceso.

   7[4] = 2 segundos aproximadamente

5. Juntar proveedores y envíos en el sitio A, restringir el resultado a los proveedores de Lon dres, proyectar el resultado sobre V# y P#, y mover el resultado al sitio B. Completar el procesamiento en el sitio B.

   T[5] = 0.1 + (100000 * 200) / 50000
   ■ 400 segundos aproximadamente (6.67 minutos)

6. Restringir las partes en el sitio B a las que son rojas y mover el resultado al sitio A. Com pletar el procesamiento en el sitio A.

```
7"[6] ■  0.1 + (10 * 200) / 50000
         0.1 segundos aproximadamente
```

La figura 20.4 resume los resultados anteriores. Surgen los siguientes puntos:

Estrategia	Técnica	Tiempo de comunicación
1	Mover p hacia A	6.67 minutos
2	Mover V y VP hacia B	1.12 horas
3	Para cada envió de Londres, revisar si la parte es roja	5.56 horas
4	Para cada parte roja, revisar si existe un proveedor de Londres	2.00 segundos
5	Mover los envíos de Londres hacia B	6.67 minutos
6	**Mover las partes rojas hacia A**	**0.10 segundos** (el mejor)

Figura 20.4 Estrategias de procesamiento de consultas distribuidas (resumen).

- Cada una de las seis estrategias representan un enfoque admisible para el problema, pero la variación en el tiempo de comunicación es enorme (la más lenta es *dos millones* de veces más lenta que la más rápida).
- La velocidad de datos y el retardo en el acceso son factores importantes para la selección de una estrategia.
- Es probable que los tiempos del cálculo y la E/S sean despreciables en comparación con el tiempo de comunicación para las peores estrategias. (Por otro lado, para las mejores estra tegias es posible que esto importe o no [20.35]. También es posible que no sea el caso en una LAN rápida.)

Además, algunas estrategias permiten el procesamiento paralelo en los dos sitios; por lo tanto, el tiempo de respuesta ante el usuario puede ser, de hecho, menor que en un sistema centralizado. Sin embargo, observe que hemos ignorado el asunto de qué sitio va a recibir el resultado final.

Administración del catálogo

En un sistema distribuido, el catálogo del sistema incluirá no solamente los datos usuales del catálogo con relación a las varrels base, vistas, autorizaciones, etcétera, sino también toda la información de control necesaria para permitir que el sistema proporcione la independencia de ubicación, fragmentación y replicación necesaria. Surge la siguiente cuestión: ¿dónde y cómo debe ser almacenado el propio catálogo? Estas son algunas posibilidades:

1. *Centralizado:* el catálogo total es almacenado exactamente una vez en un sitio central.
2. *Completamente replicado:* el catálogo total es almacenado por completo en cada uno de los sitios.
3. *Dividido:* cada sitio mantiene su propio catálogo de los objetos que están almacenados en ese sitio. El catálogo total es la unión de todos los catálogos locales disjuntos.
4. *Combinación de 1 y 3:* cada sitio mantiene su propio catálogo local, como en el punto 3; además, un único sitio central mantiene una copia unificada de todos esos catálogos locales, como en el punto 1.

Cada uno de estos enfoques tiene sus problemas. El enfoque 1 viola obviamente el objetivo de "no dependencia de un sitio central". El enfoque 2 sufre una severa pérdida de autonomía, ya que cada actualización del catálogo tiene que ser propagada a cada uno de los sitios. El enfoque 3 eleva en gran medida el costo de las operaciones que no son locales (para localizar un objeto remoto será necesario acceder, en promedio, a la mitad de los sitios). El enfoque 4 es más eficiente que el 3 (la localization de un objeto remoto requiere sólo un acceso al catálogo remoto), pero viola nuevamente el objetivo de "no dependencia de un sitio central". Por lo tanto, en la práctica ¡los sistemas generalmente no usan *ninguno* de estos cuatro enfoques! A manera de ejemplo describimos el enfoque usado en R* [20.39].

Para explicar la forma en que está estructurado el catálogo de R*, primero es necesario decir algo acerca del **nombramiento de objetos** en R*. Ahora bien, el nombramiento de objetos es un asunto importante para los sistemas distribuidos en general, ya que la posibilidad de que dos sitios distintos, *X y Y,* puedan tener un objeto, digamos una varrel base llamada A, implica que sería necesario algún mecanismo —por lo general la calificación por nombre de sitio— para "eliminar la ambigüedad" (es decir, garantizar la unicidad de nombres a nivel sistema). Sin embargo, si se exponen nombres calificados como *XA* y *YA* al usuario, se violará claramente el objetivo de independencia de ubicación. Por lo tanto, lo que se necesita es un medio para transformar los nombres conocidos por los usuarios a sus nombres correspondientes conocidos por el sistema.

Éste es el enfoque de R* para este problema. R* primero distingue entre el **nombre común** de un objeto, que es el nombre por el cual los usuarios hacen normalmente referencia al objeto (por ejemplo, en una instrucción SELECT de SQL), y su **nombre a nivel sistema,** que es un identificador interno globalmente único para el objeto. Los nombres a nivel sistema tienen cuatro componentes:

- *ID del creador* (el ID del usuario que creó el objeto);
- *ID del sitio del creador* (el ID del sitio en el cual se dio la operación CREATE);
- *Nombre local* (el nombre del objeto sin calificativos);
- *ID del sitio de nacimiento* (el ID del sitio en el cual se almacenó inicialmente el objeto).

(Los IDs de usuario son únicos dentro del sitio y los IDs del sitio son únicos al nivel global.) Por lo tanto, el nombre a nivel sistema

```
MARIO 3 NUEVAYORK . STATS @ LONDRES
```

denota un objeto, tal vez una varrel base, con el nombre local STATS, creada por el usuario Mario en el sitio de Nueva York y almacenada inicialmente* en el sitio de Londres. **Está garantizado que este nombre nunca cambiará,** ni aunque el objeto emigre a otro sitio (vea más adelante).

Como ya indicamos, los usuarios se refieren normalmente a los objetos por su *nombre común.* Un nombre común consiste en un nombre simple sin calificativos, ya sea el componente "nombre local" del nombre a nivel sistema (STATS en el ejemplo anterior) o un **sinónimo** para ese nombre a nivel sistema, definido por medio de la instrucción especial de SQL, R* CREATE SYNONYM. Éste es un ejemplo:

```
CREATE SYNONYM MSTATS FOR MARIO @ NUEVAYORK . STATS @ LONDRES ;
```

Las varrels base están almacenadas físicamente en R, tal como sucede en casi todos los sistemas que conozco.

Ahora el usuario puede decir (por ejemplo)

```
SELECT...FROM STATS... ;
```

O

```
SELECT...FROM MSTATS... ;
```

En el primer caso (al usar el nombre local), el sistema infiere el nombre a nivel sistema suponiendo todos los valores predeterminados obvios; es decir, que el objeto fue creado por este usuario, que fue creado en este sitio y que fue guardado inicialmente en este sitio. Dicho sea de paso, una consecuencia de estas suposiciones predeterminadas es que las aplicaciones antiguas del System R se ejecutarán sin cambios en R* (es decir, una vez que los datos del System R hayan sido redefinidos ante R*; recuerde que System R fue el prototipo predecesor de R*).

En el segundo caso (al usar el sinónimo), el sistema determina el nombre a nivel sistema consultando la **tabla de sinónimos** relevante. Las tablas de sinónimos pueden ser vistas como el primer componente del catálogo; cada sitio mantiene un conjunto de esas tablas para los usuarios que se sabe que están en ese sitio y transforma los sinónimos conocidos para ese usuario en los nombres a nivel sistema correspondientes.

Además de las tablas de sinónimos, cada sitio mantiene:

1. una entrada de catálogo para cada objeto **nacido** en ese sitio;
2. una entrada de catálogo para cada objeto **almacenado actualmente** en ese sitio.

Supongamos que ahora el usuario emite una solicitud que hace referencia al sinónimo MSTATS. Primero, el sistema busca el nombre a nivel sistema correspondiente en la tabla de sinónimos adecuada (una simple búsqueda local). Ahora ya sabe el sitio de nacimiento (es decir, Londres en el ejemplo) y puede consultar al catálogo de Londres (y suponemos, de manera general, que será una búsqueda remota; el primer acceso remoto). El catálogo de Londres contendrá una entrada para ese objeto gracias al punto 1 anterior. Si el objeto está todavía en Londres ya habrá sido encontrado. Sin embargo, si el objeto ha emigrado (digamos) a Los Angeles, entonces la entrada de catálogo en Londres lo dirá y por lo tanto, el sistema podrá ahora consultar al catálogo de Los Angeles (segundo acceso remoto). Y el catálogo de Los Angeles contendrá una entrada para el objeto gracias al punto 2 anterior. Por lo tanto, ha sido encontrado en, como máximo, dos accesos remotos.

Además, si el objeto emigra nuevamente, digamos a San Francisco, entonces el sistema:

- insertará una entrada en el catálogo de San Francisco;
- borrará la entrada del catálogo de Los Ángeles;
- actualizará la entrada del catálogo de Londres para que apunte a San Francisco en lugar de a Los Ángeles.

El efecto neto es que el objeto todavía puede ser encontrado en dos accesos remotos, como máximo. Y éste es un esquema completamente distribuido; no hay un sitio con catálogo central y no j hay punto alguno de falla dentro del sistema.

Observamos que el esquema de nombramiento de objetos que se usa en la propiedad de datos distribuidos del DB2 es similar, pero no idéntico, al que describimos anteriormente.

Propagación de la actualización

Como señalamos en la sección 20.3, el problema básico que existe con la replicación de datos es que una actualización a cualquier objeto lógico dado debe ser propagada a todas las copias almacenadas de ese objeto. Una dificultad inmediata es que un sitio que mantiene una copia del objeto podría no estar disponible (debido a una falla del sitio o de la red) en el momento de la actualización. Por lo tanto, la estrategia obvia para propagar inmediatamente las actualizaciones hacia todas las copias, es probablemente inaceptable, ya que implica que la actualización —y por ende la transacción— fallará si alguna de esas copias no está disponible actualmente. De hecho, en cierto sentido los datos están *menos* disponibles bajo esta estrategia que como lo estarían en el caso no replicado, y por lo tanto, debilita a una de las ventajas proclamadas para la replicación en la sección anterior.

Un esquema común para atacar este problema (aunque no es el único posible) es el llamado esquema de **copia primaria,** que funciona de la siguiente forma:

- A una copia de cada objeto replicado se le designa como copia *primaria*. Todas las demás son copias secundarias.
- Las copias primarias de diferentes objetos están en diferentes sitios (por lo tanto, éste es nuevamente un esquema distribuido).
- Decimos que las operaciones de actualización quedaron lógicamente terminadas tan pronto como se actualizó la copia primaria. Entonces, el sitio que mantiene esa copia es responsa ble de la propagación de la actualización hacia las copias secundarias en algún tiempo sub secuente. (Ese "tiempo subsecuente" debe ser previo al COMMIT, si es que se van a conservar las propiedades ACID de la transacción. Vea a continuación comentarios adicionales sobre este tema.)

Por supuesto, este esquema conduce a varios problemas adicionales propios, y la mayoría de ellos caen más allá del alcance de este libro. También observe que representa una violación al objetivo de autonomía local, debido a que una transacción ahora puede fallar debido a que alguna copia remota (primaria) de un objeto podría no estar disponible; aunque sí esté disponible una copia local.

Nota: Como mencionamos anteriormente, los requerimientos ACID del procesamiento de transacciones implican que es necesario terminar toda la propagación de la actualización antes de poder terminar la transacción relevante ("replicación síncrona"). Sin embargo, varios productos comerciales soportan actualmente una forma de replicación menos ambiciosa en la cual la propagación de la actualización se realiza en algún momento posterior (posiblemente en algún momento especificado por el usuario) y *no* necesariamente dentro del alcance de la transacción relevante ("replicación asincrona"). Además, el término *replicación* ha sido (por desgracia) más o menos usurpado por esos productos, con el resultado de que —al menos en el mercado comercial— casi siempre es tomada como si implicara que la propagación de la actualización se retarda hasta más allá del punto de confirmación de la transacción relevante (por ejemplo, vea las referencias [20.1], [20.18] y [20.21]). Por supuesto, el problema con este enfoque de propagación retrasado es que ya no es posible garantizar que la base de datos sea consistente en todo momento; además, el usuario ni siquiera puede saber si es consistente o no.

Cerramos esta subsección con un par de observaciones adicionales sobre el enfoque de la propagación retardada.

1. El concepto de replicación en un sistema con propagación de actualización retardada, puede ser visto como una aplicación de la idea de *instantáneas,* como explicamos en el capítulo 9.* Además, habría sido mejor usar un término diferente para este tipo de replicación, ya que así habríamos podido mantener el término "réplica" para que significara lo que generalmente se entiende en el discurso normal (es decir, una copia exacta). *Nota:* No pretendemos sugerir que la propagación retardada sea una mala idea; es claramente lo correcto en circunstancias adecuadas, como veremos por ejemplo en el capítulo 21. Sin embargo, el punto es que la propagación retardada significa que las "réplicas" no son réplicas verdaderas y el sistema en realidad no es un sistema de base de datos distribuida.
2. Una razón (y tal vez la principal) por la cual los productos están implementando la repli cación con propagación retardada es que la alternativa —es decir, la actualización de todas las réplicas antes del COMMIT— requiere un soporte de confirmación de dos fases (vea más adelante) que puede ser costoso en términos de rendimiento. Esta situación explica los artículos encontrados ocasionalmente en las revistas con títulos desconcertantes como "La replicación contra la confirmación de dos fases" (desconcertantes ya que superficialmente parecen estar comparando los méritos de dos cosas totalmente diferentes).

Control de la recuperación

Como explicamos en la sección 20.3, el control de la recuperación en sistemas distribuidos está basado típicamente en el protocolo de **confirmación de dos fases** (o en alguna variante). La confirmación de dos fases es necesaria en *cualquier* ambiente en el cual una transacción única puede interactuar con varios administradores de recursos autónomos, aunque es particularmente importante en un sistema distribuido debido a que los administradores de los recursos en cuestión —es decir, los DBMSs locales— están operando en sitios distintos y por lo tanto, son *muy* autónomos. Surgen los siguientes puntos:

1. El objetivo de la "no dependencia de un sitio central" indica que la función coordinadora no debe ser asignada a un sitio distinguido en la red, sino que en su lugar, debe ser realizada por sitios diferentes para transacciones diferentes. Por lo general es manejada por el sitio en el cual se inicia la transacción en cuestión; por lo tanto (en general), cada sitio debe ser capaz de actuar como sitio coordinador para algunas transacciones y como sitio participante para otras.
2. El proceso de confirmación de dos fases requiere que el coordinador se comunique con cada uno de los sitios participantes, lo cual significa más mensajes y más sobrecarga.
3. Si el sitio *Y* actúa como participante en un proceso de confirmación de dos fases coordinado por el sitio *X,* entonces el sitio *Y debe* hacer lo que le indica el sitio *X* (confirmar o deshacer, según se aplique) y esto es una pérdida (tal vez menor) de autonomía local.

Revisemos el proceso de confirmación de dos fases básico tal como lo describimos en el capítulo 14. Consulte la figura 20.5, que muestra las interacciones que suceden entre el coordinador

*Con excepción de que las instantáneas supuestamente son de sólo lectura (independientemente de las actualizaciones periódicas), mientras que algunos sistemas permiten que los usuarios actualicen las "réplicas" directamente; por ejemplo, vea la referencia [20.21]. Por supuesto, esta última posibilidad constituye una violación a la independencia de replicación.

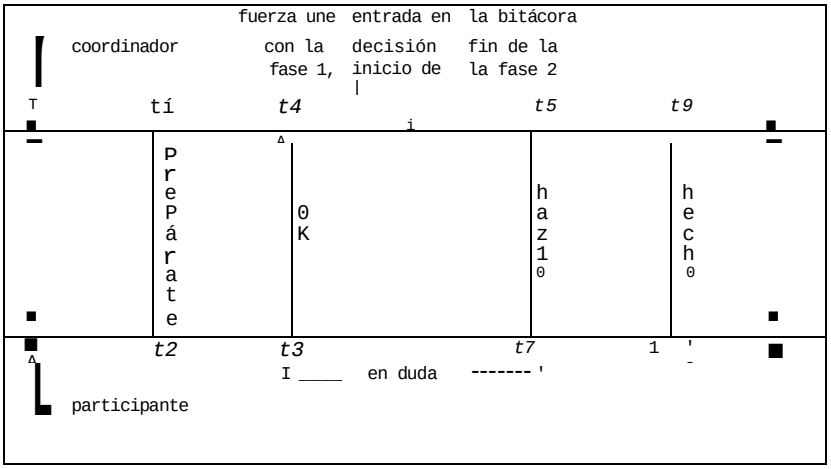


Figura 20.5 Confirmación de dos fases.

y un participante típico (el cual suponemos, por razones de simplicidad, que está en un sitio remoto). El tiempo en la figura va de izquierda a derecha (¡más o menos!). Suponemos, por razones de simplicidad, que la transacción ha solicitado un COMMIT y no un ROLLBACK. Al recibir esa solicitud de COMMIT, el coordinador realiza el siguiente proceso de dos fases.

- Da instrucciones a cada participante para que se "prepare para ir por cualquier camino" en la transacción. La figura 20.5 muestra el mensaje "prepárate", que es enviado en el tiempo *ti* y recibido por el participante en el tiempo *t2*. El participante ahora fuerza una entrada de bitácora para el agente local en su propia bitácora, y luego responde "OK" al coordinador (por supuesto, si cualquier cosa ha salido mal —en particular si ha fallado el agente local— en su lugar responde "no OK"). En la figura, esta respuesta —que por razones de simplici dad suponemos que es "OK"— es enviada en el tiempo *t3* y recibida por el coordinador en el tiempo *t4*. Observe que (como ya indicamos) el participante sufre ahora una pérdida de autonomía: *debe* hacer lo que subsecuentemente le diga el coordinador. Además, cualquier recurso bloqueado por el agente local *debe permanecer bloqueado* hasta que el participante reciba y actúe de acuerdo con la decisión del coordinador.
- Cuando el coordinador ha recibido las respuestas de todos los participantes toma su decisión (la cual será *confirmar* si todas las respuestas fueron "OK" o *deshacer* en caso contrario). Luego fuerza una entrada en su propia bitácora física, registrando esa decisión en el tiempo *t5*. Este tiempo *t5* marca la transición de la fase 1 hacia la fase 2.
- Suponemos que la decisión fue *confirmar*. Por lo tanto, el coordinador da instrucciones a cada participante para que "lo haga" (es decir, que realice el procesamiento de confirmación para el agente local). La figura 20.5 muestra que el mensaje "hazlo" es enviado en el tiempo *t6* y recibido por el participante en el tiempo *t7*. El participante confirma al agente local y envía una notificación ("hecho") de regreso al coordinador. En la figura, esa notificación es enviada en el tiempo *t8* y recibida por el coordinador en el tiempo *t9*.

- Cuando el coordinador ha recibido todas las notificaciones termina todo el proceso.

Por supuesto, en la práctica el proceso general es considerablemente más complicado de lo que acabamos de decir, ya que debemos preocuparnos por la posibilidad de que ocurra una falla en el sitio o en la red. Por ejemplo, supongamos que el sitio coordinador falla en algún tiempo í entre los tiempos *t5* y *t6.* Luego, cuando el sitio se recupera, el procedimiento de reinicio descubrirá en la bitácora que una determinada transacción estaba en la fase dos del proceso de confirmación de dos fases, y continuará el proceso de enviar los mensajes "hazlo" a todos los participantes. (Observe que el participante está "en duda" en la transacción del período que va de *t3* a í7; si el coordinador falla en el tiempo *t,* como sugerimos, ese período "en duda" puede ser bastante largo.)

Por supuesto, de forma ideal nos gustaría que el proceso de confirmación de dos fases fuera resistente a *cualquier* tipo de falla *concebible.* Por desgracia, es fácil ver que este objetivo es fundamentalmente inalcanzable; es decir, no existe ningún protocolo finito que *garantice* que todos los participantes confirmarán una transacción satisfactoria al unísono, o la desharán al unísono ante fallas arbitrarias. Pero suponga que, de manera inversa, tal protocolo existe. Sea *N* la cantidad mínima de mensajes requeridos por ese protocolo. Suponga ahora que el último de estos /V mensajes se pierde debido a una falla. Entonces ese mensaje era innecesario, lo que contradice la suposición de que *N* era mínimo, o el protocolo ahora no funciona. De cualquier forma, ésta es una contradicción a partir de la cual deducimos que no existe tal protocolo.

A pesar de este hecho deprimente, hay varias mejoras posibles para el algoritmo básico que buscan mejorar el rendimiento:

- En primer lugar, si el agente de algún sitio participante específico es de sólo lectura, ese participante puede responder "ignórame" en la fase 1 y el coordinador puede ignorar a ese participante en la fase 2.
- Segundo, si *todos* los participantes responden "ignórame" en la fase 1, es posible omitir completamente la fase 2.
- Tercero, hay dos variantes importantes sobre el esquema básico, que son *confirmar de manera presupuesta* y *deshacer de manera presupuesta* [20.15], las cuales describimos con mayor detalle en los siguientes párrafos.

En general, los esquemas para confirmar de manera presupuesta y deshacer de manera presupuesta, tienen el efecto de reducir el número de mensajes involucrados en el caso de una operación exitosa (para confirmar de manera presupuesta) o en el caso de falla (para deshacer de manera presupuesta). Para explicar los dos esquemas, observemos primero que el mecanismo básico —tal como le describimos anteriormente— requiere que el coordinador recuerde su decisión hasta que haya recibido una notificación de cada participante. Por supuesto, la razón es que si un participante tiene una falla mientras está "en duda", tendrá que consultar al coordinador durante el reinicio para descubrir cuál fue la decisión del mismo. Sin embargo, una vez que se han recibido todas las notificaciones, el coordinador sabe que todos los participantes han hecho lo que se les dijo y por lo tanto, puede "olvidar" la transacción.

Nos concentraremos ahora en **confirmar de manera presupuesta.** Bajo este esquema se requiere que los participantes notifiquen sobre los mensajes "deshacer" ("cancelar"), pero no sobre los mensajes "confirmar" ("hacer"), y el coordinador puede olvidar la transacción tan pronto como haya transmitido su decisión, *siempre y cuando* esa decisión sea "confirmar". Si falla algún participante en duda, consultará (como siempre) al coordinador durante el reinicio. Si el

coordinador aún recuerda la transacción (es decir, el coordinador sigue esperando la notificación del participante), entonces la decisión debió haber sido "deshacer"; de lo contrario, debió haber sido "confirmar".

Por supuesto, cancelar de manera presupuesta es lo contrario. Los participantes deben notificar sobre los mensajes "confirmar", pero no sobre los mensajes "cancelar", y el coordinador puede olvidar la transacción tan pronto como haya transmitido su decisión, siempre y cuando esa decisión sea "deshacer". Si falla algún participante que está en duda, entonces interrogará al coordinador durante el reinicio. Si el coordinador aún recuerda la transacción (es decir, si el coordinador sigue esperando la notificación del participante) entonces la decisión fue "confirmar"; en caso contrario fue "deshacer".

Es interesante (y hasta cierto punto, ilógico) que las acciones deshacer de manera presupuesta parezcan ser preferibles a las de confirmar de manera presupuesta (decimos "ilógico" debido a que seguramente la mayoría de las transacciones son satisfactorias y confirmar de manera presupuesta reduce la cantidad de mensajes en el caso de que todo salga bien). El problema que tiene confirmar de manera presupuesta es el siguiente. Suponga que el coordinador falla en la fase 1 (es decir, antes de que haya tomado la decisión). Al reinicio en el sitio del coordinador, la transacción será deshecha (debido a que no terminó). Posteriormente, algún participante consulta al coordinador pidiéndole su decisión con respecto a esta transacción. El coordinador no recuerda la transacción y presupone "confirmar", lo cual por supuesto es incorrecto.

Para evitar esas "confirmaciones falsas" el coordinador (si está siguiendo la confirmación presupuesta) debe forzar una entrada de bitácora en su propia bitácora física al inicio de la fase 1, dando una lista de todos los participantes en la transacción. (Si el coordinador falla ahora en la fase 1, puede transmitir "deshacer" a todos los participantes durante el reinicio.) Y esta E/S física para la bitácora del coordinador se encuentra en la ruta crítica de *cada una de las transacciones.* Por lo tanto, confirmar de manera presupuesta no es tan atractivo como pudiera parecer a primera vista. De hecho, es probablemente correcto decir que, al momento de la publicación de este libro, deshacer de manera presupuesta es el estándar *defacto* en los sistemas implementados .

Control de la concurrencia

Como explicamos en la sección 20.3, en la mayoría de los sistemas distribuidos (al igual que en la mayoría de los sistemas no distribuidos) el control de la concurrencia está basado en el bloqueo. Sin embargo, en un sistema distribuido las solicitudes para probar, poner y liberar bloqueos se convierten en *mensajes* (suponiendo que el objeto en consideración está en un sitio remoto) y los mensajes significan una sobrecarga. Por ejemplo, considere una transacción *T* que necesita actualizar un objeto para el cual existen réplicas en *n* sitios remotos. Si cada sitio es responsable de los bloqueos sobre los objetos que están almacenados en ese sitio (como será bajo la suposición de autonomía local), entonces una implementación directa requerirá al menos 5n mensajes:

n solicitudes de bloqueo,

n otorgamientos de bloqueo,

n mensajes de actualización,

n notificaciones,

n solicitudes de desbloqueo.

Por supuesto, podemos resolver fácilmente lo anterior si "encimamos" los mensajes —por ejemplo, es posible combinar los mensajes de solicitud de bloqueo y los de actualización, así como los mensajes de garantía de bloqueo y los de notificación— pero aún así, el tiempo total para la actualización seguirá siendo varias veces mayor que como sucedería en un sistema centralizado.

El enfoque usual para este problema es adoptar la estrategia de **copia primaria** que mencionamos anteriormente en la subsección "Propagación de la actualización". Para un objeto dado A, el sitio que mantiene la copia primaria de *A* manejará todas las operaciones de bloqueo que involucren a *A* (recuerde que copias primarias de objetos diferentes estarán generalmente en sitios diferentes). Bajo esta estrategia, el conjunto de todas las copias de un objeto puede ser considerado como un solo objeto para efectos de bloqueo, y la cantidad total de mensajes se reducirá de *5n* a 2ra + 3 (una solicitud de bloqueo, una garantía de bloqueo, *n* actualizaciones, *n* notificaciones y una solicitud de desbloqueo). Pero observe de nuevo que esta solución implica una (severa) pérdida de autonomía; ahora, una transacción puede fallar si no hay una copia primaria disponible, aun cuando la transacción sea de sólo lectura y tenga una copia local disponible. (Observe que no sólo las operaciones de actualización, sino también las de recuperación, necesitan bloquear la copia primaria [20.15], Por lo tanto, un desagradable efecto lateral de la estrategia de copia primaria es la reducción del rendimiento y la disponibilidad para las recuperaciones y las actualizaciones.)

Otro problema con el bloqueo en un sistema distribuido es que puede conducir a un **bloqueo mortal global.** Un bloqueo mortal global es aquél que involucra dos o más sitios. Por ejemplo (consulte la figura 20.6):

1. El agente de la transacción 72 en el sitio *X* está esperando al agente de la transacción 77 del sitio *X* para que libere un bloqueo.
2. El agente de la transacción *TI* en el sitio *X* está esperando a que termine el agente de la transacción *TI* en el sitio *Y*.

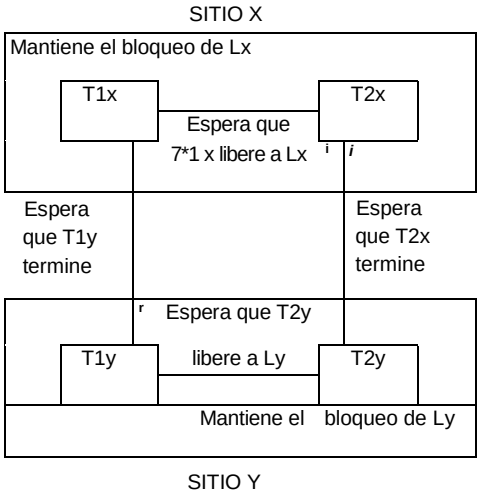


Figura 20.6 Un ejemplo de bloqueo mortal global.

3. El agente de la transacción 77 en el sitio *Y* está esperando que el agente de la transacción *T2* en el sitio *Y* libere un bloqueo.
4. El agente de la transacción *T2* en el sitio *Y* está esperando que termine el agente de la tran sacción 72 en el sitio *X: ¡bloqueo mortall*

El problema con un bloqueo mortal global como éste es que *ningún sitio puede detectarlo usando solamente información interna de ese sitio.* En otras palabras, no hay ciclos en el grafo de espera local, pero aparecerá un ciclo si se combinan esos dos grafos locales para que formen un grafo global. De aquí se desprende que la detección del bloqueo mortal global incurre en más sobrecarga de comunicaciones, debido a que requiere que los grafos locales individuales estén reunidos de alguna forma.

En los artículos de R* (vea por ejemplo la referencia [20.39]) se describe un esquema elegante (y distribuido) para detectar bloqueos mortales globales. *Nota:* Como señalamos en el capítulo 15, en la práctica no todos los sistemas detectan en realidad a los bloqueos mortales; algunos usan simplemente un mecanismo temporizador. Por razones que deben ser obvias, este comentario es particularmente cierto en los sistemas distribuidos.

20.5 SISTEMAS CLIENTE-SERVIDOR

Como mencionamos en la sección 20.1, los sistemas **cliente-servidor** pueden ser considerados como un caso especial de los sistemas distribuidos en general. Para ser más precisos, un sistema cliente-servidor es un sistema distribuido en el cual (a) algunos sitios son sitios *clientes* y algunos son *servidores,* (b) todos los datos residen en los sitios servidores, (c) todas las aplicaciones son ejecutadas en los sitios clientes y (d) "se ven las costuras" (no existe una independencia total de ubicación). Consulte la figura 20.7 (una repetición de la figura 2.5 del capítulo 2).

En este momento hay mucho interés comercial en los sistemas cliente-servidor y comparativamente poco en los sistemas distribuidos verdaderos de propósito general (aunque esto está comenzando a cambiar en cierta forma, como veremos en la siguiente sección). Continuamos creyendo que los sistemas distribuidos verdaderos representan una tendencia importante a largo plazo, y ésta es la razón por la cual en este capítulo nos concentramos en tales sistemas; pero en realidad es adecuado decir algo con relación a los sistemas cliente-servidor en particular.

Recuerde del capítulo 2, que el término "cliente-servidor" se refería principalmente a una *arquitectura* o división lógica de responsabilidades; el **cliente** es la aplicación (a quien también se conoce como *interfaz* o *parte frontal)* y el **servidor** es el DBMS (conocido también como *servicios de fondo o parte dorsal).* Sin embargo, debido precisamente a que el sistema general puede ser dividido claramente en dos partes, existe la posibilidad de ejecutar a las dos partes en máquinas diferentes. Y esta última posibilidad es tan atractiva (por muchas razones, vea el capítulo 2) que el término "cliente-servidor" ha venido aplicándose casi exclusivamente para el caso donde el cliente y el servidor están, en efecto, en máquinas diferentes.* Este uso es un poco descuidado pero común, y lo adoptaremos en lo que presentamos a continuación.

También le recordamos que son posibles muchas variaciones del tema básico:

- Varios clientes pueden compartir el mismo servidor (éste es el caso normal).
- Un solo cliente puede acceder a varios servidores. Esta última posibilidad se divide a su vez en dos casos:

*E1 término "sistema de dos capas" también se usa básicamente (por razones obvias) con el mismo significado.

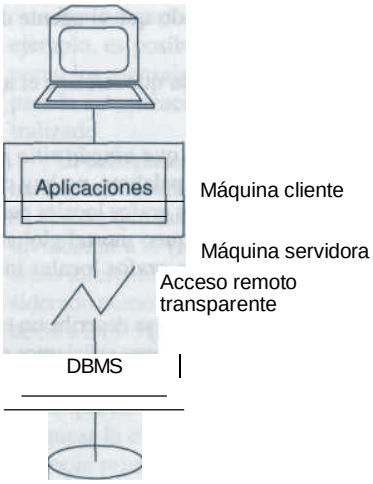


Figura 20.7 Un sistema cliente-servidor.

a. El cliente está limitado a acceder a un solo servidor a la vez; es decir, cada solicitud individual de base de datos debe ser dirigida hacia un solo servidor. No es posible combinar, dentro de una sola solicitud, datos de dos o más servidores diferentes. Además, el usuario debe saber cuál servidor en particular almacena cuáles partes de datos.

b. El cliente puede acceder a muchos servidores simultáneamente; es decir, una sola solicitud de base de datos puede combinar datos de varios servidores, lo que significa que los servidores ven al cliente como si fueran en realidad, sólo un servidor, y el usuario no tiene que saber qué servidor almacena qué partes de datos.

Pero el caso b es efectivamente un sistema verdadero de base de datos distribuida ("las costuras están ocultas"); no es el significado general del término "cliente-servidor" y lo ignoraremos en las siguientes partes.

Estándares de cliente-servidor

Hay varios estándares que son aplicables al mundo del procesamiento cliente-servidor:

- En primer lugar, ciertas características cliente-servidor están incluidas en el estándar de 1 **SQL,** SQL/92 [4.22]. Dejamos para la sección 20.7 la explicación de estas características. I

- Después, tenemos el estándar ISO de RDA **(acceso a datos remotos);** vea las referencias [20.26] y [20.27]. Una razón por la cual el RDA es importante, es porque algo muy pare cido a él ya ha sido implementado por los miembros del SAG **(grupo de acceso SQL),** que es un consorcio de fabricantes de software de base de datos comprometidos con los sistemas abiertos y la interoperabilidad. *Nota:* Para nuestros propósitos, no vale la pena preocuparse por distinguir entre las versiones del RDA de ISO y SAG; usaremos simplemente el nom bre "RDA" para referimos a ellas en forma genérica.

  El RDA pretende definir **formatos y protocolos** para las comunicaciones cliente-servidor. Supone (a) que el cliente expresa solicitudes de base de datos en una forma estándar de SQL (en esencia, un subconjunto del estándar de SQL/92) y también (b) que el servidor soporta un catálogo estándar (también como está definido en el estándar de SQL/92). Luego define formatos de representación específicos para pasar mensajes (solicitudes SQL, datos y resultados, así como información de diagnóstico) entre el cliente y el servidor.

- El tercero y último estándar que mencionamos aquí es la DRDA **(arquitectura de bases de datos relaciónales distribuidas)** de IBM [20.25] (la cual es un estándar *de hecho* y no *de ley).* La DRDA y el RDA tienen objetivos similares, pero la DRDA difiere con respecto al RDA en varios aspectos importantes; en particular, tiende a reflejar sus orígenes en IBM. Por ejemplo, la DRDA no supone que el cliente esté usando una versión estándar de SQL, sino que en cambio permite cualquier dialecto de SQL. Una consecuencia es (posiblemente) un mejor rendimiento, debido a que el cliente puede explotar determinadas características del servidor; por otro lado, presenta un deterioro en la portabilidad, precisamente debido a que estas características específicas del servidor están expuestas ante el cliente (es decir, el cliente tiene que saber con qué tipo de servidor se está comunicando). En forma similar, la DRDA no da por hecho ninguna estructura de un catálogo específico del servidor. Los formatos y protocolos de la DRDA son bastantes diferentes con respecto a los del RDA (esencialmente, la DRDA está basada en las arquitecturas y estándares propios de IBM y en cambio, el RDA está basado en estándares internacionales que no son específicos de un fabricante).

Algunos detalles adicionales sobre el RDA y la DRDA están fuera del alcance de este libro; vea las referencias [20.23] y [20.30], para algunos análisis y comparaciones.

Programación de aplicaciones cliente-servidor

Hemos dicho que un sistema cliente-servidor es un caso especial de los sistemas distribuidos en general. Como sugerimos en la introducción de esta sección, podemos pensar en un sistema cliente-servidor como un sistema distribuido en el cual todas las solicitudes se originan en un sitio y todo el procesamiento se realiza en otro (dando por hecho, por razones de simplicidad, que sólo hay un sitio cliente y un sitio servidor). *Nota:* Por supuesto, bajo esta definición sencilla, el sitio cliente en realidad no es "un sitio de sistema de base de datos por derecho propio" y por lo tanto, el sistema contradice la definición de un sistema de base de datos distribuida de propósito general que dimos en la sección 20.2. (Es posible que el sitio cliente tenga sus propias bases de datos locales, pero estas bases de datos no juegan un papel principal en el arreglo cliente-servidor como tal.)

Entonces, el enfoque cliente-servidor tiene determinadas implicaciones para la programación de aplicaciones (como también sucede en los sistemas distribuidos en general). Uno de

los puntos más importantes ya lo hemos tratado en nuestra explicación del objetivo número 7 (el procesamiento de consultas distribuidas) de la sección 20.3; es decir, el hecho de que los sistemas relaciónales son, por definición y diseño, sistemas en el **nivel de conjunto.** En un sistema cliente-servidor (y también en los sistemas distribuidos en general) es más importante que nunca que el programador de aplicaciones *no* "use al servidor sólo como un método de acceso" y escriba código en el nivel de registro. En su lugar, debe integrar la mayor cantidad de funcionalidad posible en las solicitudes al nivel de conjunto; de no hacerlo así, disminuirá el rendimiento debido a la cantidad de mensajes involucrados. (En términos de SQL, lo anterior implica que hay que *evitar los cursores* tanto como sea posible; es decir, evitar los ciclos FETCH y las formas CURRENT de UPDATE y DELETE. Vea el capítulo 4.)

La cantidad de mensajes entre cliente y servidor puede ser reducida todavía más si el sistema proporciona algún tipo de mecanismo de **procedimiento almacenado.** Un procedimiento almacenado es básicamente un programa precompilado que está *almacenado en el sitio servidor* (y que es conocido por el servidor). Es llamado desde el cliente mediante una RPC **(llamada a procedimiento remoto).** Por lo tanto, la penalización en el rendimiento —asociada con el procesamiento en el nivel de registro— puede ser cancelada (en parte) con la creación de un procedimiento almacenado adecuado que haga ese procesamiento directamente en el sitio servidor.

Nota: Aunque es algo tangencial al tema del procesamiento cliente-servidor como tal, debemos señalar que el mejoramiento en el rendimiento no es la única ventaja de los procedimientos almacenados. Otras incluyen:

- Dichos procedimientos pueden ser usados para ocultar ante el usuario una diversidad de de talles específicos del DBMS o de la base de datos, y por lo tanto, proporcionan un mayor grado de independencia de datos que el que podría obtenerse de otra forma.
- Un procedimiento almacenado puede ser compartido por muchos clientes.
- La optimización puede ser realizada en el momento de crear el procedimiento almacenado, en lugar de hacerlo en tiempo de ejecución. (Por supuesto, esta ventaja se aplica solamente a los sistemas que normalmente hacen la optimización en tiempo de ejecución.)
- Los procedimientos almacenados pueden proporcionar mayor seguridad. Por ejemplo, un usuario dado puede estar autorizado para llamar a un procedimiento dado, pero no para ope rar directamente sobre los datos a los cuales accede ese procedimiento.

Una desventaja es que diferentes fabricantes proporcionan propiedades muy diferentes en esta área, a pesar del hecho de que, como mencionamos en el capítulo 4, el estándar de SQL fue extendido en 1996 para que incluyera algún soporte para los procedimientos almacenados (bajo el nombre de *módulos almacenados persistentes* o PSM).

20.6 INDEPENDENCIA DE DBMS

Ahora regresamos a nuestra explicación de los doce objetivos para los sistemas de bases de datos distribuidos en general. Recordará que el último de esos objetivos fue la *independencia de DBMS.* Como explicamos en la breve exposición de este objetivo en la sección 20.3, la suposición de homogeneidad estricta es demasiado discutible; todo lo que en realidad necesitamos es que los DBMSs que están en sitios diferentes soporten la misma interfaz. Como dijimos en esa sección: si, por ejemplo, Ingres y Oracle soportan el estándar oficial de SQL —¡ni más ni menos! — debería

ser posible hacer que se comportaran como socios igualitarios en un sistema distribuido heterogéneo (además, dicha posibilidad es en primer lugar uno de los argumentos que generalmente se dan a favor del estándar de SQL). Comenzaremos considerando esta posibilidad en detalle. *Nota:* Basamos nuestra exposición en Ingres y Oracle específicamente, sólo para que las cosas sean un poco más concretas. Por supuesto, los conceptos son de aplicación general.

Pasarela

Suponga que tenemos dos sitios, *XyY,* que ejecutan Ingres y Oracle respectivamente, y suponga que algún usuario *U* que está en el sitio *X* desea ver una sola base de datos distribuida que incluya datos de la base de datos Ingres que está en el sitio *X* y de la base de datos Oracle que está en el sitio *Y.* Por definición, el usuario *U* es un usuario de Ingres y por lo tanto, la base de datos distribuida debe ser una base de datos Ingres para el usuario. Por lo tanto, la responsabilidad de proporcionar el soporte necesario es de Ingres y no de Oracle. ¿En qué debe consistir ese soporte? En principio, el soporte es bastante directo. Ingres debe proporcionar un programa especial — por lo general llamado **pasarela**— cuyo efecto es "hacer que Oracle se parezca a Ingres". Consulte la figura 20.8.* La pasarela puede ser ejecutada en el sitio de Ingres o en el de Oracle o (como lo sugiere la figura) en algún sitio especial propio entre los dos; sin importar dónde se ejecute, debe proporcionar con claridad al menos todas las funciones de la siguiente lista. (Observe que varias de estas funciones presentan problemas de implementación de naturaleza no muy trivial. Sin embargo, los estándares RDA y DRDA, que tratamos en la sección 20.5, atacan algunos de estos problemas.)

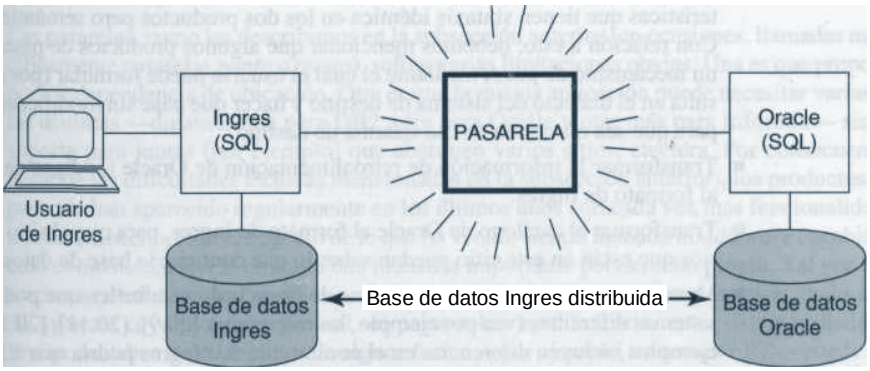


Figura 20.8 Una pasarela hipotética proporcionada por Ingres para Oracle.

*E1 término "sistema de tres capas" en ocasiones es utilizado (por razones obvias) para referirse al tipo de arreglo ilustrado en la figura, así como para otras configuraciones de software que, de manera similar, involucran tres componentes (vea en particular la explicación de "middleware" en la siguiente subsección).

Implementar los protocolos para intercambiar información entre Ingres y Oracle, lo cual implica (entre otras cosas) transformar el formato de mensaje en el cual son enviadas las instrucciones fuente de SQL desde Ingres hacia el formato esperado por Oracle, y transformar el formato de mensaje en el cual son enviados los resultados desde Oracle hacia el formato esperado por Ingres.

Proporcionar una posibilidad de "servidor relacional" para Oracle (equivalente al procesador SQL interactivo que ya se encuentra en la mayoría de los productos SQL). En otras palabras, la pasarela debe ser capaz de ejecutar cualquier instrucción SQL no planeada en la base de datos Oracle. Para tener la posibilidad de proporcionar esta función, la pasarela tendrá que usar el soporte de **SQL dinámico** o (lo más probable) una interfaz en el nivel de llamada (como **SQL/CLI** u **ODBC)** en el sitio Oracle. Vea el capítulo 4. *Nota:* En forma alterna, la pasarela puede hacer uso directo del procesador SQL interactivo que ya proporciona Oracle.

Hacer transformaciones entre los tipos de datos de Oracle e Ingres. Este problema incluye una variedad de subproblemas que tienen que ver con situaciones tales como las diferencias en el procesador (por ejemplo, diferentes longitudes de palabra de máquina), las diferencias en el código de caracteres (las comparaciones de cadenas de caracteres y las solicitudes ORDER BY pueden dar resultados inesperados), las diferencias en el formato de punto flotante (un área notoriamente problemática), las diferencias en el soporte para fecha y hora (hasta donde conozco, no existen dos DBMS que proporcionen actualmente soporte idéntico en esta área), etcétera. Vea la referencia [20.15] para más explicaciones sobre estos temas.

Transformar el dialecto de SQL de Ingres al dialecto de Oracle; ya que, de hecho, ni Ingres ni Oracle soportan exactamente el estándar de SQL, "ni más ni menos". En realidad, ambos soportan determinadas características que el otro no soporta, y también hay algunas características que tienen sintaxis idéntica en los dos productos pero semántica diferente. *Nota:* Con relación a esto, debemos mencionar que algunos productos de pasarela proporcionan un mecanismo *de paso,* mediante el cual el usuario puede formular (por ejemplo) una consulta en el dialecto del sistema de destino y hacer que pase sin modificación por la pasarela para que sea ejecutada en ese sistema de destino.

Transformar la información de retroalimentación de Oracle (códigos de retorno, etcétera) al formato de Ingres.

Transformar el catálogo de Oracle al formato de Ingres, para que el sitio Ingres y los usuarios que están en este sitio puedan saber lo que contiene la base de datos Oracle.

Manejar una diversidad de problemas de **desacoplo semántico** que podrían suceder entre sistemas diferentes (vea por ejemplo, las referencias [20.9], [20.11], [20.16] y [20.38]). Los **i** ejemplos incluyen diferencias en el nombramiento (Ingres podría usar EMP# mientras que Oracle, EMPNUM); diferencias en tipos de datos (Ingres podría usar cadenas de caracteres i mientras que Oracle, números); diferencias en unidades (Ingres podría usar centímetros mientras que Oracle, pulgadas); diferencias en la representación lógica de la información (Ingres podría omitir tupias mientras que Oracle, usar nulos) y mucho, mucho más.

Servir como participante en (la variante de Ingres para) el protocolo de confirmación de dos fases (suponiendo que se permita que las transacciones de Ingres realicen actualizaciones en la base de datos Oracle). La posibilidad de que la pasarela realice esta función, dependerá de las propiedades proporcionadas por el administrador de transacciones en el sitio |

Oracle. Vale la pena señalar que al momento de la publicación de este libro, los administradores de transacciones comerciales (con algunas excepciones) *no* proporcionaban lo que se necesita en este aspecto; en particular, la posibilidad para que un programa de aplicación dé instrucciones al administrador de transacciones para que "esté listo para terminar" (contrario a las instrucciones de terminar; es decir, confirmar o deshacer incondicionalmente).

- Asegurarse de que los datos en el sitio Oracle que Ingres requiere como bloqueados, estén en realidad bloqueados cuando Ingres los necesite. De nuevo, la capacidad de la pasarela para realizar esta función dependerá presuntamente de que las arquitecturas de bloqueo de Oracle y de Ingres coincidan o no.

Hasta ahora hemos tratado la independencia de DBMS sólo en el contexto de sistemas relaciónales. ¿Qué hay acerca de los sistemas no relaciónales?; es decir, ¿qué hay acerca de la posibilidad de incluir un sitio no relacional en un sistema distribuido que, por lo demás, es relaciona]? Por ejemplo, ¿sería posible proporcionar acceso a un sitio IMS desde un sitio Ingres u Oracle? Una vez más, tal característica sería muy buena en la práctica, tomando en cuenta la enorme cantidad de datos que residen actualmente en IMS y otros sistemas anteriores a los relaciónales.* Pero ¿puede hacerse?

Si la pregunta significa "¿puede hacerse al 100 por ciento?" —lo que significa "¿pueden hacerse accesibles todos los datos no relaciónales desde una interfaz relacional y realizar todas las operaciones relaciónales sobre esos datos?"— entonces la respuesta es un *no* categórico, por las razones explicadas en detalle en la referencia [20.16]. Pero si la pregunta significa, "¿es posible proporcionar algún nivel útil de funcionalidad?", entonces la respuesta es obviamente *sí*. Este no es el lugar para entrar en detalles, pero vea las referencias [20.14] a [20.16] para explicaciones adicionales.

Middleware para acceso a datos

Las pasarelas, como las describimos en la subsección anterior (en ocasiones, llamadas más específicamente pasarelas *punto a punto),* sufren varias limitaciones obvias. Una es que proporcionan poca independencia de ubicación. Otra es que la misma aplicación puede necesitar varias pasarelas distintas —digamos una para DB2, otra para Oracle y otra más para Informix— sin ningún soporte para juntas (por ejemplo) que abarquen varios sitios, etcétera. Por consecuencia (y a pesar de las dificultades técnicas mencionadas en la subsección anterior), los productos del tipo pasarela han aparecido regularmente en los últimos años con cada vez más funcionalidad sofisticada. De hecho, todo el negocio de lo que ha venido siendo llamado *middleware* (también conocido como *mediadores)* es ahora una industria importante por derecho propio. Tal vez no cause sorpresa que el término "middleware" no esté definido de manera muy precisa; cualquier pieza de software cuyo propósito general sea pasar sobre las diferencias entre sistemas distintos, que se supone deben trabajar juntos de alguna forma —por ejemplo, un monitor PT— puede ser vista razonablemente como "middleware" [20.3]. Sin embargo, aquí nos concentramos en lo que podría ser llamado middleware de *acceso a datos.* Ejemplos de dichos productos incluyen a Cohera de Cohera Inc., DataJoiner de IBM Corp. y OmniConnect e InfoHub de Sybase Inc. A manera de ejemplo, describimos brevemente el producto DataJoiner [20.7].

---

*La idea convencional es que tal vez el 85 por ciento de los datos de negocios residen todavía en estos sistemas (es decir, sistemas de bases de datos anteriores a los relaciónales, e incluso sistemas de archivos) y hay muy pocos indicios de que los clientes moverán próximamente sus datos hacia sistemas más recientes.

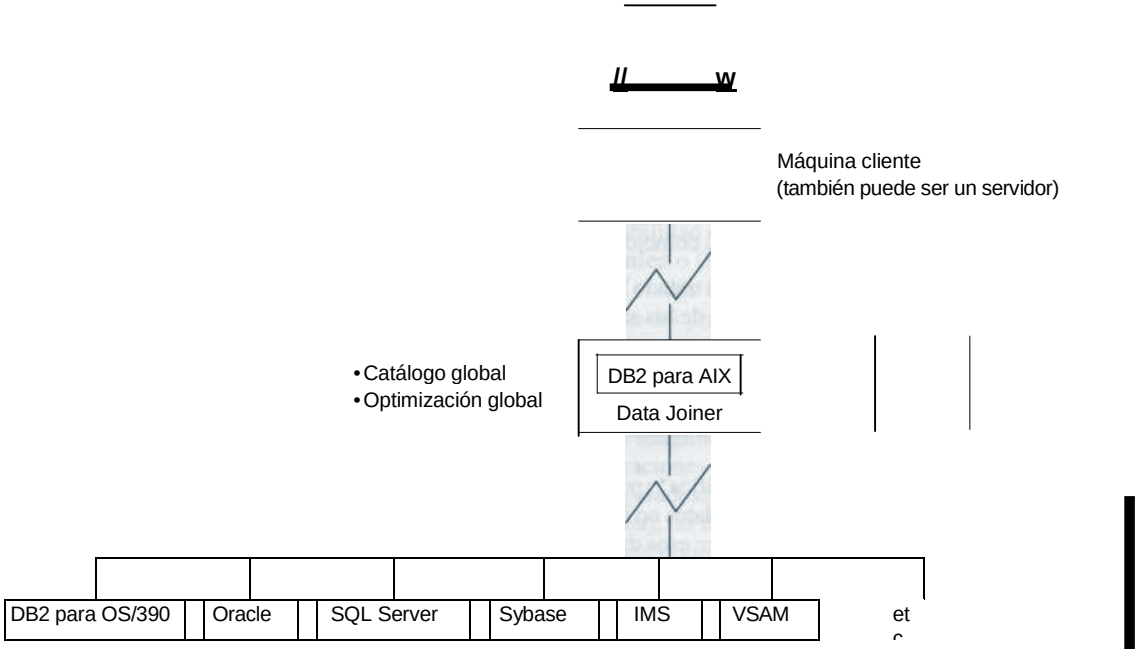


Figura **20.9** El DataJoiner como middleware de acceso a datos.

Hay varias formas diferentes para caracterizar a DataJoiner (vea la figura 20.9). Desde el punto de vista de un cliente individual, se parece a un servidor normal de bases de datos (es decir, un DBMS), ya que guarda datos, soporta consultas SQL (estilo DB2), proporciona un catálogo, realiza una optimización de consultas, etcétera (de hecho, la parte medular de DataJoiner es la versión AIX del producto DBMS DB2 de IBM). Sin embargo, la mayoría de los datos no está guardada en el sitio DataJoiner (aunque tiene esa capacidad), sino en varios otros sitios ocultos tras bambalinas que están bajo control de una variedad de otros DBMSs (o incluso de administradores de archivos como VSAM). Por lo tanto, DataJoiner proporciona al usuario una base de datos virtual que es la unión de todos esos almacenes de datos "tras bambalinas"; permite que las consultas* abarquen esos almacenes de datos, y usa su conocimiento de las posibilidades de los sistemas que están tras bambalinas (y de las características de la red) para determinar los planes de consulta "globalmente óptimos".

*Énfasis sobre las "consultas"; las posibilidades de actualización están un tanto limitadas, en especial (pero no únicamente) cuando el sistema oculto tras bambalinas es, digamos, IMS u otro sistema que no sea SQL (de nuevo, vea la referencia [20.16]). Al momento de la publicación de este libro, DataJoiner en realidad soportaba transacciones de actualización (con confirmación de dos fases) sólo a través de sitios DB2, Oracle, Sybase e Informix.

Nota: DataJoiner también incluye la posibilidad de emular ciertas características SQL del DB2 en sistemas que no soportan directamente esas características. Un ejemplo podría ser la opción WITH HOLD en una declaración de cursor (vea el capítulo 14).

Ahora, el sistema tal como lo hemos descrito hasta este momento, todavía no es un sistema de base de datos distribuido completo, debido a que los diversos sitios que están ocultos no están conscientes de la existencia de los demás (es decir, no pueden ser considerados como socios igualitarios en una empresa cooperativa). Sin embargo, cuando se añade cualquier sitio "tras bambalinas", este nuevo sitio también puede comportarse como cliente y por lo tanto, emitir consultas por medio de DataJoiner que accedan a cualquiera o a todos los demás sitios. Entonces, el sistema general constituye lo que a veces se llama un sistema **federado,** también conocido como sistema de **bases de datos múltiples** [20.19]. Un sistema federado es un sistema distribuido, usualmente heterogéneo, con una autonomía local cercana a la total; en un sistema de éstos, las transacciones puramente locales son administradas por el DBMS local pero las transacciones globales son un asunto diferente [20.8].

De manera interna, DataJoiner incluye un componente *controlador* —en efecto una pasarela punto a punto en el sentido de la subsección anterior— para cada uno de los sistemas "tras bambalinas". (Por lo general, estos controladores utilizan ODBC para acceder al sistema remoto.) También mantiene un *catálogo global,* el cual se usa (entre otras cosas) para que le diga lo que debe hacer cuando encuentre desacoplos semánticos entre esos sistemas.

Hacemos notar que productos como DataJoiner pueden ser útiles para los fabricantes de software de terceras partes, quienes pueden desarrollar herramientas genéricas (por ejemplo, generadores de informes, paquetes estadísticos, etcétera) sin tener que preocuparse demasiado por las diferencias entre los productos DBMS distintos en los que supuestamente se ejecutarán esas herramientas.

Una última palabra

De manera clara, existen problemas importantes al tratar de proporcionar la independencia total del DBMS, aun cuando todos los DBMS participantes son sistemas SQL. Sin embargo, la ganancia potencial es inmensa aunque las soluciones no lleguen a ser perfectas; por esta razón, hay disponibles varios productos middleware de acceso a datos y es seguro que aparecerán más en un futuro cercano. Pero tenga presente que las soluciones serán necesariamente algo menos que perfectas, aunque los fabricantes digan lo contrario. Que tenga cuidado cuando compre.

20.7 PROPIEDADES DE SQL

SQL no proporciona actualmente soporte alguno para los sistemas de bases de datos distribuidas verdaderos. Por supuesto, no se *requiere* ningún soporte en el área de manipulación de datos; la cuestión es que en una base de datos distribuida (por lo que se refiere al usuario), las posibilidades de manipulación de datos deben permanecer sin cambios. Sin embargo, *sí se* requieren [20.15] operaciones de definición de datos tales como FRAGMENT, REPLICATE, etcétera, aunque actualmente no las proporciona.

Por otro lado, SQL soporta determinadas posibilidades cliente-servidor, incluyendo en particular las operaciones **CONNECT y DISCONNECT** para establecer e interrumpir conexiones cliente-servidor. De hecho, una aplicación SQL *debe* ejecutar una operación CONNECT para

conectarse a un servidor antes de que pueda emitir alguna solicitud de base de datos (aunque ese CONNECT podría ser implícito). Una vez que se ha establecido la conexión, la aplicación —es decir, el cliente— puede emitir solicitudes SQL en la forma normal, y el procesamiento necesario de base de datos será realizado por el servidor.

SQL también permite que un cliente que ya está conectado a algún servidor, se conecte a otro. El establecimiento de esa segunda conexión ocasiona que la primera permanezca **latente;** las solicitudes SQL posteriores, son procesadas por el segundo servidor hasta el momento en que el diente (a) regresa al servidor anterior (por medio de otra operación nueva SET CONNECTION) o (b) se conecta a otro servidor, lo cual ocasiona que la segunda conexión quede también latente (y así sucesivamente). En otras palabras, en cualquier momento dado, un cliente puede tener una conexión **activa** y cualquier cantidad de conexiones **latentes,** y todas las solicitudes de base de datos de ese cliente son dirigidas hacia, y procesadas por, el servidor que está en la conexión activa.

Nota: El estándar de SQL también permite (aunque no requiere) la implementación para soportar las *transacciones multiservidor.* Es decir, el cliente puede cambiar de un servidor a otro en la mitad de una transacción, para que parte de la transacción sea ejecutada en un servidor y la otra parte en otro. Observe en particular que si se permite que las transacciones de *actualización* abarquen servidores en esta forma, la implementación debe soportar, presuntamente, algún tipo de confirmación de dos fases para proporcionar la atomicidad de transacción que requiere el estándar.

Por último, cada una de las conexiones establecidas por un cliente dado (ya sea la que está activa actualmente o la latente) deben ser interrumpidas tarde o temprano por medio de una operación DISCONNECT adecuada (aunque ese DISCONNECT, al igual que el CONNECT correspondiente, puede estar implícito en los casos simples).

Para información adicional consulte el estándar de SQL [4.22] o el tratamiento tutorial que está en la referencia [4.19].

20.8 RESUMEN

En este capítulo hemos presentado una breve explicación de los sistemas de bases de datos distribuidas. Usamos los **"doce objetivos"** para los sistemas de bases de datos distribuidas [20.14] como base para estructurar la explicación, aunque enfatizamos nuevamente que no todos esos objetivos serán relevantes en todas las situaciones. También analizamos brevemente determinados problemas técnicos que se presentan en las áreas de **procesamiento de consultas, administración del catálogo, propagación de la actualización, control de la recuperación y control de la concurrencia.** En particular, explicamos lo que está involucrado al tratar de satisfacer el objetivo de **independencia de DBMS** (la explicación de las **pasarelas** y el **middleware de acceso a datos** de la sección 20.6). También profundizamos en el procesamiento **cliente-servidor,** que puede ser considerado como un caso especial del procesamiento distribuido en general y que actualmente es muy popular en el mercado. En particular, resumimos los aspectos de SQL que son importantes para el procesamiento cliente-servidor, y enfatizamos el punto de que los usuarios deben **evitar el código en el nivel de registro** (las operaciones de cursor, en términos de SQL). También describimos brevemente el concepto de **procedimientos almacenados y llamadas a procedimientos remotos.**

Nota: Un problema que no tratamos en lo absoluto es el del **diseño de base de datos** (físico) para los sistemas distribuidos. De hecho, aunque ignoremos la posibilidad de fragmentación o

replicación, el problema de decidir qué datos deben estar almacenados en cuáles sitios —el llamado **problema de reparto**— es notoriamente difícil [20.33]. El soporte para la fragmentación y la replicación sólo sirve para complicar más las cosas.

Otro punto que vale la pena mencionar es que determinados sistemas de computadora, llamados *paralelos en forma masiva,* están comenzando a hacer sentir su presencia en el mercado (vea los comentarios a la referencia 17.58 del capítulo 17). Por lo general, dichos sistemas consisten en una gran cantidad de procesadores independientes, conectados por medio de un bus de alta velocidad; cada procesador tiene su propia memoria principal y sus propias unidades de disco y ejecuta su propia copia de software DBMS, y la base de datos completa está repartida en todo el conjunto de unidades de disco. En otras palabras, dicho sistema consiste esencialmente en ¡un sistema de base de datos distribuida "en una caja"!; todas las cuestiones que hemos estado tratando en el presente capítulo con relación (por ejemplo) a las estrategias de procesamiento de consultas, confirmación de dos fases, bloqueos mortales globales, etcétera, son importantes.

Como conclusión, comentamos que en conjunto, los "doce objetivos" de las bases de datos distribuidas (o posiblemente algún subconjunto de ellos que incluye al menos a los números 4, 5,6 y 8) parecen ser equivalentes a la regla de "independencia de distribución" de Codd para los DBMSs relaciónales [9.5]. Para referencia, aquí describimos esa regla:

- *Independencia de distribución* (Codd): "Un DBMS relacional tiene independencia de distribución... [y esto significa que] el DBMS tiene un sublenguaje de datos que permite que los programas de aplicación y las actividades terminales permanezcan lógicamente intactos:
  1. Cuando la distribución de datos es introducida por primera vez (si el DBMS instalado originalmente administra solamente datos no distribuidos).
  2. Cuando los datos son redistribuidos (si el DBMS administra datos distribuidos)".

Por último, observe que (como mencionamos anteriormente en el capítulo) los objetivos 4 a 6 y 9 a 12 —es decir, todos los objetivos que incluyen en su nombre la palabra "independencia"— pueden ser considerados como extensiones a la noción familiar de independencia de datos, en la forma en que se aplica este concepto a los ambientes distribuidos. Como tales, todos ellos se traducen en una **protección para la inversión en la aplicación.**

EJERCICIOS

20.1 Defina la independencia de ubicación, la independencia de fragmentación y la independencia de replicación.

20.2 ¿Por qué son relaciónales, casi invariablemente, los sistemas de bases de datos distribuidas?

20.3 ¿Cuáles son las ventajas de los sistemas distribuidos? ¿Cuáles son las desventajas?

20.4 Explique los siguientes términos:

estrategia de actualización de copia primaria
estrategia de bloqueo de copia primaria
bloqueo mortal global confirmación de dos
fases optimización global

20.5 Describa el esquema de nombramiento de objetos de R*.

20.6 Las implementaciones exitosas de una pasarela punto a punto dependen de la conciliación de las diferencias de interfaz entre los dos DBMSs involucrados (entre otras muchas cosas). Tome dos sistemas SQL cualesquiera con los que esté familiarizado e identifique tantas diferencias de interfaz entre ellos como pueda. Considere tanto las diferencias sintácticas como las semánticas.

20.7 Investigue cualquier sistema cliente-servidor que tenga disponible. ¿Soporta ese sistema las ope raciones CONNECT Y DISCONNECT explícitas? ¿Soporta SET CONNECTION o cualquier otra operación de "tipo de conexión"? ¿Soporta transacciones entre varios servidores? ¿Soporta la confir mación de dos fases? ¿Qué formatos y protocolos usa para la comunicación cliente-servidor? ¿Qué ambientes de red soporta? ¿Qué plataformas de hardware de cliente y de servidor soporta? ¿Qué pla taformas de software (sistemas operativos, DBMSs) soporta?

20.8 Investigue cualquier DBMS SQL que tenga disponible. ¿Soporta ese DBMS los procedimientos almacenados? De ser así, ¿cómo son creados? ¿Cómo se les llama? ¿En qué lenguaje están escritos? ¿Soportan completamente a SQL? ¿Soportan la ramificación condicional (IF-THEN-ELSE)? ¿So portan ciclos? ¿Cómo regresan los resultados al cliente? ¿Puede un procedimiento almacenado llamar a otro? ¿En un sitio diferente? ¿El procedimiento almacenado es ejecutado como parte de la transac ción que lo llama?

REFERENCIAS Y BIBLIOGRAFÍA

20.1 Todd Anderson, Yuri Breitbart, Henry F. Korth y Avishai Wool: "Replication, Consistency, and Practicality: Are These Mutually Exclusive?", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

Este artículo describe tres esquemas para la replicación asincrona (llamada aquí *perezosa)* que garantizan la atomicidad de la transacción y la seriabilidad global sin usar la confirmación de dos fases, y reporta un estudio de simulación de su rendimiento comparativo. El primer esquema es el bloqueo global, tal como es propuesto en la referencia [20.21]. Los otros dos —uno de ellos pesimista y el otro optimista— utilizan un *grafo de replicación.* El artículo concluye que los es-quemas de grafo de replicación sobrepasan por lo general al esquema de bloqueo "por un amplio margen".

20.2 David Bell y Jane Grimson: *Distributed Database Systems.* Reading, Mass.: Addison-Wesley (1992).

Es uno de varios libros de texto sobre sistemas distribuidos (otros dos son las referencias [20.10] y [20.31]). Una característica notable de este libro es la inclusión de un amplio caso de estudio que involucra una red para el cuidado de la salud. También es un poco más pragmático que los otros dos.

20.3 Philip A. Bernstein: "Middleware: A Model for Distributed System Services", *CACM 39,* No. 2 (febrero, 1996).

"Clasifica diversos tipos de middleware, describe sus propiedades y explica su evolución, proporcionando un modelo conceptual para la comprensión del software de sistemas distribuidos actuales y futuros" (tomado del resumen).

20.4 Philip A. Bernstein, James B. Rothnie, Jr. y David W. Shipman (eds.): *Tutorial: Distributed Data Base Management.* IEEE Computer Society, 5855 Naples Plaza, Suite 301, Long Beach, Calif. 90803 (1978).

Es un conjunto de artículos de diversas fuentes, agrupados bajo los siguientes temas:

1. Panorama de la administración de bases de datos relaciónales.
2. Panorama de la administración de bases de datos distribuidas.
3. Enfoques del procesamiento de consultas distribuidas.
4. Enfoques del control de la concurrencia distribuida.
5. Enfoques de la confiabilidad de bases de datos distribuidas.

20.5 Philip A. Bernstein *et al.:* "Query Processing in a System for Distributed Databases (SDD-1)", *ACM TODS 6,* No. 4 (diciembre, 1981).

Vea el comentario a la referencia [20.34],

20.6 Philip A. Bernstein, David W. Shipman y James B. Rothnie, Jr: "Concurrency Control in a System for Distributed Databases (SDD-1)", *ACM TODS 5,* No. 1 (marzo, 1980).

Vea el comentario a la referencia [20.34].

20.7 Charles J. Bontempo y C. M. Saracco: "Data Access Middleware: Seeking out the Middle Ground", *InfoDB 9,* No. 4 (agosto, 1995).

Un tutorial útil con énfasis en DataJoiner de IBM (aunque también se mencionan otros productos).

20.8 Yuri Breitbart, Héctor Garcia-Molina y Avi Silberschatz: "Overview of Multi-Database Transaction Management", *The VLDB Journal 1,* No. 2 (octubre, 1992).

20.9 M. W. Bright, A. R. Hurson y S. Pakzad: "Automated Resolution of Semantic Heterogeneity in Multi-Databases", *ACM TODS 19,* No. 2 (junio, 1994).

20.10 Stefano Ceri y Giuseppe Pelagatti: *Distributed Databases: Principles and Systems.* Nueva York, N.Y.: McGraw-Hill (1984).

20.11 William W. Cohen: "Integration of Heterogeneous Databases without Common Domains Using Queries Based on Textual Similarity", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

Describe un enfoque de lo que a veces se llama "problemas de correo chatarra"; es decir, reconocer cuándo dos cadenas de texto distintas, digamos "AT&T Bell Labs" y "AT&T Research", se refieren al mismo objeto (un tipo particular de desacoplo semántico, por supuesto). El enfoque involucra el razonamiento sobre la *similitud* de tales cadenas "tal como es medida usando el modelo de espacio de vectores adoptado comúnmente en la recuperación de información estadística". De acuerdo con el artículo, el enfoque es mucho más rápido que los "métodos de inferencia ingenuos" y de hecho, es sorprendentemente preciso.

20.12 D. Daniels *et ai:* "An Introduction to Distributed Query Compilation in R*", en H.-J. Schneider (ed.), *Distributed Data Bases:* Proc. 2nd Int. Symposium on Distributed Data Bases (septiembre, 1982). Nueva York, N.Y.: North-Holland (1982).

Vea el comentario a la referencia [20.39].

20.13 C. J. Date: "Distributed Databases", Capítulo 7 de *An Introduction to Database Systems: Volu me ¡I.* Reading, Mass.: Addison-Wesley (1983).

Partes del presente capítulo están basadas en esta publicación anterior.

20.14 C. **J.** Date: "What Is a Distributed Database System?", en *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Es el artículo que presentó los "doce objetivos" para los sistemas distribuidos (la sección 20.3 está modelada muy directamente sobre este artículo). Como mencionamos en el cuerpo del capítulo, el objetivo de la *autonomía local* no es alcanzable al 100 por ciento; hay determinadas situaciones que involucran necesariamente compromisos de alguna forma sobre ese objetivo. Aquí resumimos esas situaciones para efectos de referencia:

- Normalmente, no es posible acceder directamente a fragmentos individuales de una varrel fragmentada, ni siquiera desde el sitio en el que están almacenados.
- Normalmente, no es posible acceder directamente a copias individuales de una varrel replica da (o fragmentada), ni siquiera desde el sitio en el que están almacenadas.
- Sea *P* la copia primaria de alguna varrel *R* replicada (o fragmentada), y sea que *P* está alma cenada en el sitio *X*. Entonces, todo sitio que acceda a *R* es dependiente del sitio *X*, aunque otra copia de *R* esté de hecho almacenada en el sitio en cuestión.
- No es posible acceder a una varrel que participa en una restricción de integridad de varios sitios, para efectos de actualización, dentro del contexto local del sitio en el que está almace nada, sino sólo dentro del contexto de la base de datos distribuida en el que está definida la restricción.
- Un sitio que está actuando como participante en un proceso de confirmación de dos fases, debe obrar de acuerdo a las decisiones (es decir confirmar o deshacer) del sitio coordinador correspondiente.

20.15 C. J. Date: "Distributed Database: A Closer Look", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991*. Reading, Mass.: Addison-Wesley (1992).

Es la continuación de la referencia [20.14] que trata la mayoría de los doce objetivos con mucha mayor profundidad (aunque todavía en estilo de tutorial).

20.16 C. J. Date: "Why Is It So Difficult to Provide a Relational Interface to IMS?", en *Relational Database: Selected Writings*. Reading, Mass.: Addison-Wesley (1986).

20.17 R. Epstein, M. Stonebraker y E. Wong: "Distributed Query Processing in a Relational Data Base System", Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data, Austin, Tx. (mayo-junio, 1978).

Vea el comentario a la referencia [20.36].

20.18 Rob Goldring: "A Discussion of Relational Database Replication Technology", *InfoDB 8*, No. 1 (primavera, 1994).

Es un buen panorama general de la replicación asincrona.

20.19 John Grant, Witold Litwin, Nick Roussopoulos y Timos Sellis: "Query Languages for Rela tional Multi-Databases", *The VLDB Journal 2*, No. 2 (abril, 1993).

Propone extensiones al álgebra relacional y al cálculo relacional para manejar sistemas con varias bases de datos. Trata asuntos de optimización y muestra que toda expresión algebraica multirrelacional tiene un equivalente en el cálculo multirrelacional ("la parte inversa de este teorema es un problema de investigación interesante").

20.20 J. N. Gray: "A Discussion of Distributed Systems", Proc. Congresso AICA 79, Bari, Italia (oc tubre, 1979). También está disponible como IBM Research Report RJ2699 (septiembre, 1979).

Es un panorama general y un tutorial superficiales pero buenos.

20.21 Jim Gray, Pat Helland, Patrick O'Neil y Dennis Shasha: "The Dangers of Replication and a Solution", Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (junio, 1996).

"La replicación transaccional de actualizar en cualquier lugar, cualquier momento y cualquier forma tiene un comportamiento inestable conforme aumenta la carga de trabajo... Se propone un nuevo algoritmo que permite a las aplicaciones móviles (desconectadas) proponer transacciones de actualización tentativas que posteriormente son aplicadas a una copia maestra" (tomado del resumen, cambiando ligeramente las palabras).

20.22 Ramesh Gupta, Jayant Haritsa y Krithi Ramamritham: "Revisiting Commit Processing in Dis tributed Database Systems", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Propone un nuevo protocolo de confirmación distribuido, llamado OPT, que (a) es fácil de implementar, (b) puede coexistir con los protocolos tradicionales y (c) "proporciona el mayor rendimiento efectivo de transacciones para una variedad de cargas de trabajo y configuraciones de sistemas".

20.23 Richard D. Hackathorn: "Interoperability: DRDA or RDA?", *InfoDB 6*, No. 2 (otoño, 1991).

20.24 Michael Hammer y David Shipman: "Reliability Mechanisms for SDD-1: A System for Dis tributed Databases", *ACM TODS* 5, No. 4 (diciembre, 1980).

Vea el comentario a la referencia [20.34].

20.25 IBM Corporation: *Distributed Relational Database Architecture Reference.* IBM Form No. SC26-4651.

La DRDA de IBM define cuatro niveles de funcionalidad en las bases de datos distribuidas de la siguiente forma:

1. Solicitud remota.
2. Unidad de trabajo remota.
3. Unidad de trabajo distribuida.
4. Solicitud distribuida.

Debido a que estos términos han llegado a ser estándares *defacto* en la industria (o al menos en algunas partes de ella), aquí los explicamos brevemente. *Nota:* "Solicitud" y "unidad de trabajo" son los términos de IBM para los términos *instrucción* y *transacción*, respectivamente, de SQL.

1. **Solicitud remota** significa que una aplicación que está en un sitio *X* puede enviar una ins trucción SQL individual a algún sitio remoto Fpara su ejecución. Esa solicitud es ejecutada *y confirmada* (o deshecha) completamente en el sitio *Y*. La aplicación original que está en el sitio *X* puede enviar posteriormente una solicitud al sitio *Y* (o posiblemente, a otro sitio Z), sin tomar en cuenta si la primera solicitud fue satisfactoria o no.
2. **Unidad de trabajo remota** (abreviada RUW) significa que una aplicación que está en un sitio *X* puede enviar todas las solicitudes de bases de datos en una "unidad de trabajo" (es decir, transacción) dada hacia algún sitio remoto Fpara su ejecución. Por lo tanto, el procesa miento de base de datos para la transacción es ejecutado completamente en el sitio remoto *Y*, pero el sitio local *X* decide si la transacción va a ser confirmada o deshecha. *Nota:* RUW es en efecto un procesamiento cliente-servidor con un solo servidor.
3. **Unidad de trabajo distribuida** (abreviada DUW) significa que una aplicación que está en un sitio *X* puede enviar algunas o todas las solicitudes de bases de datos en una unidad de trabajo (transacción) dada hacia uno o más sitios remotos *Y*, Z, ..., para su ejecución. Por lo tanto, el procesamiento de base de datos para la transacción, es repartido en varios sitios en general; cada solicitud individual sigue siendo ejecutada completamente en un solo sitio, pero solicitudes diferentes pueden ser ejecutadas en sitios diferentes. Sin embargo, el sitio *X* sigue siendo el sitio coordinador, es decir, el sitio que decide si la transacción se confirma o deshace. *Nota:* DUW es en efecto un procesamiento cliente-servidor con muchos servidores.
4. **Solicitud distribuida** es el único de los cuatro niveles que se acerca a lo que consideramos comúnmente como soporte a una base de datos distribuida verdadera. La solicitud distribuida

significa todo lo que significa la unidad de trabajo distribuida y *además,* permite que las solicitudes a bases de datos individuales (instrucciones SQL) abarquen varios sitios; por ejemplo, una solicitud que se origina en el sitio *X* puede pedir que una junta o unión sea ejecutada entre una tabla que está en el sitio *Y* y otra tabla que está en el sitio Z. Observe que es solamente en este nivel donde podemos decir que el sistema está proporcionando independencia de ubicación genuina; en los tres casos anteriores los usuarios deben tener algún conocimiento con respecto a la ubicación física de los datos.

20.26 International Organization for Standardization (ISO): *Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 1: Generic Model, Service, and Protocol (Draft International Standard).* Documento ISO DIS 9579-1 (marzo, 1990).

20.27 International Organization for Standardization (ISO): *Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 2: SQL Specialization (Draft International Stan dard).* Documento ISO DIS 9579-2 (febrero, 1990).

20.28 B. G. Lindsay *et al:* "Notes on Distributed Databases", IBM Research Report RJ2571 (julio, 1979).

Este artículo (escrito por algunos de los miembros originales del equipo del R*) está dividido en cinco capítulos:

1. Datos replicados.
2. Autorización y vistas.
3. Introducción a la administración de transacciones distribuidas.
4. Facilidades de recuperación.
5. Iniciación, emigración y terminación de transacciones.

El capítulo 1 trata el problema de la propagación de la actualización. El capítulo 2 se refiere casi totalmente a la autorización en un sistema *no* distribuido (al estilo de System R), con excepción de unos cuantos comentarios al final. El capítulo 3 considera la iniciación y terminación de transacciones, el control de la concurrencia y el control de la recuperación, todo brevemente. El capítulo 4 está dedicado al tema de la recuperación en el caso *no* distribuido (nuevamente). Por último, el capítulo 5 trata la administración de transacciones distribuidas con algún detalle y en particular, da una presentación muy cuidadosa de la confirmación de dos fases.

20.29 C. Mohan y B. G. Lindsay: "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", Proc. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distribu ted Computing (1983).

Vea el comentario a la referencia [20.39].

20.30 Scott Newman y Jim Gray: "Which Way to Remote SQL?", *DBP&D 4,* No. 12 (diciembre, 1991).

20.31 M. Tamer Özsu y Patrick Valduriez: *Principles of Distributed Database Systems* (2a. edición). Englewood Cliffs, N.J.: Prentice-Hall (1999).

20.32 Martin Rennhackkamp: "Mobile Database Replication", *DBMS 10,* No. 11 (octubre, 1997).

La combinación de computadoras baratas y altamente portátiles junto con las comunicaciones inalámbricas, permiten un nuevo tipo de sistema de base de datos distribuida con sus propios be neficios especiales; pero también (por supuesto) sus propios problemas especiales. En particular, los datos en un sistema de éstos pueden ser replicados literalmente en miles de "sitios", pero esos sitios son móviles, con frecuencia están fuera de línea, sus características operacionales son muy diferentes con respecto a los sitios más convencionales (por ejemplo los costos de comunicación deben tomar en cuenta el uso de baterías y el tiempo de conexión), etcétera. La investigación sobre estos sistemas es comparativamente nueva (las referencias [20.1] y [20.21] son importantes); este breve artículo resalta algunos de los conceptos y preocupaciones principales.

20.33 James B. Rothnie, Jr. y Nathan Goodman: "A Survey of Research and Development in Distri buted Database Management", Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japón (octubre, 1977). También fue publicado en la referencia [20.4].

Es un primer estudio muy útil. El tema es tratado dentro de las siguientes secciones:

1. Sincronización de transacciones de actualización.
2. Procesamiento de consultas distribuidas.
3. Manejo de fallas de los componentes.
4. Administración del directorio.
5. Diseño de la base de datos.

El último de éstos se refiere al problema del diseño *físico,* lo que llamamos el problema de *reparto* en la sección 20.8.

20.34 J. **B.** Rothnie, Jr., *et al:* "Introduction to a System for Distributed Databases (SDD-1)", *ACM TODS 5,* No. 1 (marzo, 1980).

Las referencias [20.5], [20.6], [20.24], [20.34] y [20.40] están relacionadas con el prototipo distribuido inicial del SDD-1, que se ejecutaba en un conjunto de máquinas PDP-10 de DEC conectadas entre sí por medio de Arpanet. Este prototipo proporcionó independencia de ubicación completa, fragmentación y replicación. Aquí proporcionamos unos cuantos comentarios sobre aspectos seleccionados del sistema.

Procesamiento de consultas. El optimizador de consultas del SDD-1 (vea las referencias [20.5] y [20.40]) hace un amplio uso del operador de *sentijunta,* tal como lo describimos en el capítulo 6. La ventaja del uso de semijuntas en el procesamiento de consultas distribuidas es que puede tener el efecto de reducir la cantidad de datos que se envían a través de la red. Por ejemplo, supongamos que la varrel V de proveedores está almacenada en el sitio A, que la varrel VP de envíos está almacenada en el sitio B y que la consulta es simplemente "juntar proveedores y envíos". En lugar de enviar todo V a B (digamos), podemos hacer lo siguiente:

- Calcular la proyección (TEMPI) de VP sobre V# en B.
- Enviar TEMPI a A.
- Calcular la semijunta (TEMP2) de TEMPI y V sobre V# en A.
- Enviar TEMP2 a B.
- Calcular la semijunta de TEMP2 y VP sobre V# en B. El resultado es la respuesta a la con sulta original.

Este procedimiento reducirá obviamente la cantidad total de movimiento de datos a través de la red si y sólo si

tamaño (TEMP1) + *tamaño* (TEMP2) < *tamaño* (V)

donde el "tamaño" de una relación es la cardinalidad de esa relación multiplicada por la anchura de una tupia individual (digamos en bits). Por lo tanto, el optimizador necesita claramente tener la posibilidad de estimar el tamaño de los resultados intermedios, como TEMPI y TEMP2.

Propagación de la actualización. El algoritmo de propagación de la actualización de SDD-1 es "propagar inmediatamente" (no hay noción de una copia primaria).

Control de la concurrencia. El control de la concurrencia está basado en una técnica llamada **marca de tiempo,** en lugar del bloqueo; el objetivo es evitar la sobrecarga de mensajes asociada con el bloqueo, pero ¡el precio parece ser que, en realidad, no hay mucha concurrencia! Los

detalles están fuera del alcance de este libro (aunque el comentario de la referencia [15.3] describe brevemente la idea básica). Para mayor información vea las referencias [20.6] o [20.13].

Control de la recuperación. La recuperación está basada en un protocolo de confirmación de *cuatro* fases; el propósito es hacer al proceso más resistente que el protocolo convencional de confirmación de dos fases ante una falla en el sitio coordinador; pero desgraciadamente, también hace que el proceso sea considerablemente más complejo. Los detalles están (de nuevo) fuera del alcance de este libro.

Catálogo. El catálogo es administrado como si se tratara de datos ordinarios del usuario; puede estar arbitrariamente fragmentado y los fragmentos pueden estar arbitrariamente replicados y distribuidos, al igual que cualquier otro dato. Las ventajas de este enfoque son obvias. Por supuesto, la desventaja es que debido a que el sistema no tiene un conocimiento previo de la ubicación de ninguna parte del catálogo, es necesario mantener un catálogo de nivel más alto —el **localizador del directorio**— ¡para proporcionar exactamente esa información! El localizador del directorio está completamente replicado (es decir, una copia está guardada en cada uno de los sitios).

20.35 P. G. Selinger y M. E. Adiba: "Access Path Selection in Distributed Data Base Management Systems", en S. M. Deen y P. Hammersley (eds.), Proc. Int. Conf. on Data Bases, Aberdeen, Escocia (julio, 1980). Londres, Inglaterra: Heyden and Sons Ltd. (1980).

Vea el comentario a la referencia [20.39].

20.36 M. **R.** Stonebraker y E. J. Neuhold: "A Distributed Data Base Version of Ingres", Proc. 2nd Berkeley Conf. on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory (mayo, 1977).

Las referencias [20.17], [20.36] y [20.37] están relacionadas con el prototipo de Ingres distribuido. El Ingres distribuido consiste en varias copias del Ingres universitario ejecutadas en varias máquinas PDP-11 de DEC conectadas entre sí. Soporta independencia de ubicación (al igual que el SDD-1 y el R*); también soporta la fragmentación de datos (por medio de la restricción pero no de la proyección), con independencia de fragmentación, y la replicación de datos para tales fragmentos, con independencia de replicación. A diferencia del SDD-1 y de R*, el Ingres distribuido no da **necesariamente** por hecho que la red de comunicaciones es lenta; por el contrario, está diseñado para manejar tanto las redes "lentas" (larga distancia) como redes locales (es decir, comparativamente rápidas) (el optimizador entiende la diferencia entre los dos casos). El algoritmo de optimización de consultas es en esencia una extensión de la estrategia de descomposición de Ingres que describimos en el capítulo 17 de este libro, y es descrita en detalle dentro de la referencia [20.17].

El Ingres distribuido proporciona dos algoritmos de propagación de la actualización: un algoritmo de "rendimiento" que funciona actualizando una copia primaria y luego regresando el control a la transacción (dejando que las actualizaciones propagadas se realicen en paralelo mediante un conjunto de procesadores esclavos), y un algoritmo "confiable" que actualiza inmediatamente todas las copias (vea la referencia [20.37]). En ambos casos, el control de la concurrencia está basado en el bloqueo. La recuperación está basada en la confirmación de dos fases con mejoras.

Con relación al catálogo, el Ingres distribuido usa una combinación de replicación completa para determinadas partes del catálogo —en esencia, para las partes que contienen una descripción lógica de las varrels visibles para el usuario y una descripción de la manera en que esas varrels están fragmentadas—junto con entradas del catálogo completamente locales para otras partes, como las que describen las estructuras de almacenamiento físico local, las estadísticas locales de la base de datos (usadas por el optimizador) y las restricciones de seguridad e integridad.

20.37 M. R. Stonebraker: "Concurrency Control and Consistency of Multiple Copies in Distributed Ingres", *IEEE Transactions on Software Engineering 5*, No. 3 (mayo, 1979).

Vea el comentario a la referencia [20.36].

20.38 Wen-Syan Li y Chris Clifton: "Semantic Integration in Heterogeneous Databases Using Neural Networks", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

20.39 R. Williams *etal:* "R*: An Overview of the Architecture", en P. Scheuermann (ed.), *Improving Database Usability and Responsiveness.* Nueva York, N.Y.: Academic Press (1982). También está disponible como IBM Research Report RJ3325 (diciembre, 1981).

Las referencias [20.12], [20.29], [20.35] y [20.39] están relacionadas con R*, la versión distribuida del prototipo original del System R. R* proporciona independencia de ubicación, pero no la fragmentación ni la replicacion, y por lo tanto, tampoco la independencia de fragmentación ni de replicacion. Por la misma razón, no se presenta la cuestión de la propagación de la actualización. El control de concurrencia está basado en el bloqueo (observe que hay una sola copia de cualquier objeto a bloquear; tampoco se presenta la cuestión de la copia primaria). La recuperación está basada en la confirmación de dos fases, con mejoras.

20.40 Eugene Wong: "Retrieving Dispersed Data from SDD-1: A System for Distributed Data bases", en la referencia [20.4].

Vea el comentario a la referencia [20.34].

20.41 C. T. Yu y C. C. Chang: "Distributed Query Processing", *ACM Comp. Surv. 16,* No. 4 (di ciembre, 1984).

Es un estudio tutorial de técnicas para la optimizacion de consultas en sistemas distribuidos. Incluye una amplia bibliografía.

CAPITULO 21

Apoyo para la toma de decisiones

21.1 INTRODUCCIÓN

Nota: David McGoveran fue el autor original de este capítulo.

Los **sistemas de apoyo para la toma de decisiones** son sistemas que ayudan en el análisis de información de negocios. Su propósito es ayudar a la administración para que "marque tendencias, señale problemas y tome... decisiones inteligentes" [21.7]. Los orígenes de dichos sistemas —investigación de operaciones, teorías de administración científicas o basadas en comportamiento, y control de procesos estadísticos— se remontan a finales de los años cuarenta y principios de los cincuenta, mucho antes de que las computadoras existieran. La idea básica era, y por supuesto sigue siendo, recolectar datos *operacionales* del negocio (vea el capítulo 1) y reducirlos a una forma que pudiera ser usada para analizar el comportamiento del mismo y modificarlos de una manera inteligente. Por supuesto, por razones muy obvias, la reducción de los datos en esos primeros días era casi mínima, involucraba generalmente un poco más que producir simples informes de resúmenes.

A finales de los años sesenta y principios de los setenta, los investigadores de Harvard y del MU comenzaron a promover el uso de computadoras para que ayudaran en el proceso de toma de decisiones [21.23]. Al principio, dicho uso estuvo limitado (principalmente) a la automatización de las tareas de generación de informes, aunque en ocasiones también se proporcionaron herramientas analíticas algo rudimentarias [21.2], [21.3], [21.6] [21.26]. Esos primeros sistemas de cómputo fueron conocidos inicialmente como **sistemas de decisiones para la administración,** y posteriormente también se les llamó **sistemas de información para la administración.** Sin embargo, preferimos el término más moderno *sistemas de apoyo para la toma de decisiones,* ya que en buena medida, *todos* los sistemas de información —que incluyen, por ejemplo, a los sistemas OLTP (procesamiento de transacciones en línea)— pueden o deben ser considerados como "sistemas de información para la administración" (a fin de cuentas, todos están involucrados con y afectan la administración de los negocios). En las siguientes partes, nos mantendremos con el término más moderno.

Los años setenta también vieron el desarrollo de varios **lenguajes de consulta,** y alrededor de dichos lenguajes se construyeron varios sistemas personalizados de apoyo para la toma de decisiones (desarrollados por las propias empresas). Fueron implementados usando generadores de informes, tales como el RPG, o productos para la recuperación de datos, tales como Focus, Datatrieve y NOMAD. Estos sistemas fueron los primeros que permitieron a los usuarios finales debidamente entrenados acceder directamente a los almacenes de datos de la computadora; es decir, permitieron que tales usuarios formularan consultas relacionadas con el negocio basándose en esos almacenes de datos, y ejecutaran esas consultas directamente sin tener que esperar ayuda del departamento de procesamiento de datos.

Por supuesto, los almacenes de datos que acabamos de mencionar eran en su mayoría archivos simples; la mayoría de los datos de negocios en ese tiempo se guardaban en dichos archivos, o posi-

blemente en bases de datos no relaciónales (los sistemas relaciónales todavía se encontraban en el campo de la investigación). Por lo general, incluso en este último caso, los datos tenían que ser extraídos de la base de datos y copiados hacia archivos antes de que un sistema de apoyo para la toma de decisiones pudiera tener acceso a ellos. No fue sino hasta principios de los ochenta cuando se empezaron a usar las bases de datos relaciónales, en lugar de los archivos simples para apoyo a la toma de decisiones. De hecho, el apoyo para la toma de decisiones, las consultas *ad hoc* y la elaboración de informes estuvieron entre los primeros usos comerciales de la tecnología relacional.

Aunque en la actualidad los productos SQL están disponibles ampliamente, la idea del **procesamiento de extracciones** —es decir, copiar los datos del ambiente operacional a otro ambiente— sigue siendo muy importante; permite que los usuarios operen de cualquier forma sobre los datos extraídos sin interferir con el ambiente operacional. Y por supuesto, la razón para realizar tales extracciones es, muy a menudo, el apoyo para la toma de decisiones.

Debe quedar claro en la breve historia anterior, que el apoyo para la toma de decisiones en realidad no es parte de la tecnología de base de datos por sí misma. En su lugar, es un *uso* de esa tecnología (aunque muy importante) o para ser más precisos, son varios usos distintos, pero entrelazados. Los usos en cuestión reciben los nombres de *data warehouse, data mart, almacén de datos operacionales, OLAP (procesamiento analítico en línea), bases de datos multidimensionales* y *minería de datos* (entre otros). Trataremos todos estos temas en las secciones que vienen a continuación.

Precaución: Le comentamos inmediatamente que algo que tienen en común todas estas áreas es que ¡rara vez siguen buenos principios de diseño lógico! La práctica del apoyo para la toma de decisiones lamentablemente no es tan científica como debería; en realidad, a menudo se hace a la medida. En particular, tiende a ser manejada mucho más por consideraciones físicas que por lógicas (y además, la diferencia entre los asuntos físicos y los lógicos está a menudo muy difusa en el ambiente de apoyo para la toma de decisiones). En parte por estas razones, en este capítulo usamos SQL y no **Tutorial D** como base para nuestros ejemplos, y usamos la terminología SQL "más general" de filas, columnas y tablas en lugar de nuestra terminología preferida de tupias, atributos y valores, y variables de relación (varrels). También, usamos los términos *esquema lógico* y *esquema físico* como sinónimos para lo que en el capítulo 2 llamamos el esquema *conceptual* y el esquema *interno*, respectivamente.

Por lo tanto, el plan del capítulo es el siguiente. En la sección 21.2 tratamos determinados aspectos del apoyo para la toma de decisiones que han motivado ciertas prácticas de diseño que creemos que están un poco equivocadas. La sección 21.3 describe nuestro propio enfoque preferido para lidiar con esos aspectos. Luego, la sección 21.4 examina el tema de la preparación de datos (es decir, el proceso de poner los datos operacionales en una forma que pueda ser útil para los propósitos del apoyo para la toma de decisiones); también considera brevemente los "almacenes de datos operacionales". La sección 21.5 trata los data warehouses, los data marts y los "esquemas dimensionales". La sección 21.6 explora el OLAP (procesamiento analítico en línea) y las bases de datos multidimensionales. La sección 21.7 trata la minería de datos. La sección 21.8 presenta un resumen.

ASPECTOS DEL APOYO PARA LA TOMA DE DECISIONES

Las bases de datos de apoyo para la toma de decisiones muestran determinadas características especiales, de las cuales sobresale ésta: **la base de datos es principalmente** (aunque no totalmente) **de sólo lectura.** Por lo general, la actualización que se da está limitada a operaciones de *carga* o *actualizaciones* periódicas (y esas operaciones están dominadas, a su vez, por INSERTS —los DELETEs se realizan muy ocasionalmente y los UPDATEs casi nunca). *Nota:* En algunas ocasiones se hacen actualizaciones en determinadas tablas de trabajo auxiliares, pero el proceso

de apoyo para la toma de decisiones normal, casi nunca actualiza la propia base de datos de apoyo para la toma de decisiones.

También vale la pena señalar las siguientes características adicionales de las bases de datos de apoyo para la toma de decisiones (en la sección 21.3 regresaremos a este punto para ampliarlo). Observe que las tres primeras son de naturaleza lógica y las tres últimas son físicas.

- Se tiende a usar las columnas en combinación.
- Por lo general, no preocupa la integridad (se supone que los datos son correctos cuando se cargan por primera vez y no son actualizados posteriormente).
- Las claves incluyen frecuentemente un componente temporal (vea el capítulo 22).
- La base de datos tiende a ser grande (especialmente cuando se acumulan los detalles de las transacciones* de negocios a lo largo del tiempo, y con frecuencia así sucede).
- La base de datos tiende a estar muy indexada.
- La base de datos involucra frecuentemente varios tipos de *redundancia controlada* (vea el capítulo 1).

Las consultas de apoyo para la toma de decisiones presentan también características especiales y en particular, tienden a ser bastante *complejas.* Éstos son algunos de los tipos de complejidades que pueden presentarse:

- *Complejidad de expresiones lógicas:* Las consultas de apoyo para la toma de decisiones a menudo involucran expresiones complejas en la cláusula WHERE; las cuales son difíciles de escribir, difíciles de comprender y difíciles de manejar adecuadamente por el sistema. (En par ticular, los optimizadores clásicos tienden a ser inadecuados, debido a que están diseñados para evaluar solamente una cantidad limitada de estrategias de acceso.) Un problema común es que las consultas involucran al *tiempo;* por lo general, los sistemas actuales no proporcio nan un buen soporte para, por ejemplo, consultas que preguntan por las filas que tienen un valor máximo de marca de tiempo dentro de un periodo especificado (de nuevo, vea el capí tulo 22). Si hay alguna junta involucrada, dichas consultas llegan rápidamente a ser muy com plejas. Por supuesto, en todos esos casos el resultado neto es un bajo rendimiento.
- *Complejidad de juntas:* Las consultas de apoyo para la toma de decisiones requieren frecuen temente acceso a muchas clases de hechos. Por consecuencia, en una base de datos diseñada adecuadamente —es decir completamente normalizada— dichas consultas involucran, por lo general, a muchas juntas. Desafortunadamente, la tecnología para el procesamiento de juntas nunca ha logrado mantenerse al paso ante las demandas siempre crecientes de las consultas de apoyo para la toma de decisiones, t Por lo tanto, los diseñadores a menudo optan por *desnorma-*

---

*Aquí, y a lo largo de este capítulo, distinguimos entre las transacciones de negocios (por ejemplo, la venta de un producto) y las transacciones en el sentido de la parte IV de este libro, y usaremos siempre el calificador "negocios" cuando en realidad queramos referirnos a una transacción de negocios (a menos que el contexto haga obvio el significado).

*E1 escritor (McGoveran), mientras trabajaba en los primeros sistemas de apoyo para la toma de decisiones en 1981, observó que una junta de tres tablas de tamaño moderado podría llevarse fácilmente varias horas. Por lo general, las juntas de cuatro o seis tablas eran consideradas demasiado costosas. En la actualidad, las juntas de seis a diez tablas muy grandes son comunes, y por lo general la tecnología funciona bien. Sin embargo, todavía es fácil (y no inusual) generar consultas que junten más tablas de las que la tecnología puede manejar razonablemente. Las consultas que juntan más de doce tablas pueden llegar a ser rápidamente una aventura, ¡y aún así, es común encontrar requerimientos para dichas consultas!

lizar la base de datos "prejuntando" determinadas tablas. Sin embargo, como vimos en el capítulo 12 (en la sección 12.5), este enfoque rara vez es exitoso, ocasiona generalmente tantos problemas como los que resuelve. Además, el deseo de evitar juntas también puede conducir al uso ineficiente de las operaciones relaciónales, recuperando una gran cantidad de datos y realizando el procesamiento de juntas dentro de la aplicación en lugar de hacerlo en el DBMS.

- *Complejidad defunción:* Las consultas de apoyo para la toma de decisiones involucran fre cuentemente funciones estadísticas y matemáticas. Pocos productos soportan tales funciones. Por consecuencia, a menudo es necesario dividir una consulta en una secuencia de otras con sultas más pequeñas, las cuales luego son ejecutadas en forma intercalada con procedimientos escritos por el usuario que calculan las funciones deseadas. Este enfoque tiene la consecuen cia desafortunada de que tal vez sea necesario recuperar grandes cantidades de datos; por su puesto también hace que la consulta general sea mucho más difícil de escribir y comprender.
- *Complejidad analítica:* Las preguntas de negocios rara vez son respondidas con una sola consulta. No sólo es difícil para los usuarios escribir consultas de gran complejidad, sino que las limitaciones que tienen las implementaciones de SQL pueden impedir el procesa miento de una de estas consultas. Una forma de reducir la complejidad de dichas consultas es (de nuevo) dividirla en una serie de otras consultas más pequeñas y guardar los resulta dos intermedios en tablas auxiliares.

Todas las características anteriores, tanto las de las bases de datos de apoyo para la toma de decisiones como las de las consultas de apoyo para la toma de decisiones, dan pie a un fuerte énfasis sobre el *diseño para el rendimiento,* en especial el rendimiento de la inserción por lotes y la recuperación *ad hoc.* Sin embargo, nuestra posición (en la cual ahondaremos en la siguiente sección) es que esta situación sólo debe afectar al diseño físico de la base de datos y no al diseño lógico. Sin embargo, por desgracia (como señalamos en la sección 21.1) los fabricantes y usuarios de los sistemas de apoyo para la toma de decisiones a menudo fallan al distinguir adecuadamente entre los aspectos lógicos y los físicos;* de hecho, con frecuencia olvidan por completo el diseño lógico. Como consecuencia, los intentos para manejar las diversas características que explicamos anteriormente tienden a ser *ad hoc* y conducen frecuentemente a dificultades insuperables al tratar de balancear los requerimientos de corrección, mantenimiento, rendimiento, escalabilidad y utilidad.

21.3 DISEÑO DE BASES DE DATOS DE APOYO PARA LA TOMA DE DECISIONES

Como afirmamos anteriormente en este libro (en particular en la introducción a la parte III), nuestra posición es que el diseño de base de datos siempre debe ser realizado en al menos dos etapas, primero la lógica y luego la física:

a. El diseño lógico debe ser realizado primero. En esta etapa, el enfoque está en la *corrección relacional:* las tablas deben representar relaciones adecuadas y por lo tanto garantizar que las operaciones relaciónales funcionen tal como se indica y no produzcan resultados sor-

*Los especialistas de data warehouses y de OLAP tienden a ser especialmente culpables en esto; argumentan frecuentemente que el diseño relacional es simplemente "erróneo" para apoyar a la toma de decisiones, diciendo que el modelo relacional es incapaz de representar los datos y que debe ser dejado de lado. Tales argumentos casi siempre se deben a una falla para distinguir entre el modelo relacional y su implementación física.

prendentes. Los dominios (tipos) deben ser especificados, sus columnas definidas y las dependencias entre columnas (DFs, etcétera) deben ser identificadas. A partir de esta información es posible continuar con la normalización y definir las restricciones de integridad.

b. Segundo, el diseño físico debe surgir a partir del diseño lógico. Por supuesto, en esta etapa el punto de atención está puesto sobre *la eficiencia y el rendimiento del almacenamiento.* En principio es permisible cualquier acomodo físico de los datos, siempre y cuando exista una transformación que conserve la información entre los esquemas lógico y físico, y que pueda ser expresada en el álgebra relacional [2.5]. Observe en particular, que la existencia de una transformación de éstas implica que existen vistas relaciónales del esquema físico que lo hacen ver similar al esquema lógico y *viceversa*.

Por supuesto, el esquema lógico puede cambiar posteriormente (por ejemplo, para acomodar nuevos tipos de datos o nuevas —o recientemente descubiertas— dependencias) y tal cambio requerirá naturalmente un cambio correspondiente en el esquema físico. Dicha posibilidad no nos concierne aquí. Lo que nos concierne es la posibilidad de que el esquema físico pueda cambiar mientras no cambie el esquema lógico. Por ejemplo, supongamos que la junta de las tablas VP (envíos) y P (partes) es, por mucho, el patrón de acceso dominante. Entonces tal vez queramos "prejuntar" las tablas VP y P al nivel físico y por lo tanto, reducir los costos de E/S y de junta. Sin embargo, *el esquema lógico debe permanecer sin cambio,* si es que se desea lograr la independencia física de datos. (Por supuesto, el optimizador de consultas necesitará estar consciente de la existencia de la "prejunta" guardada y usarla adecuadamente si queremos obtener los beneficios de rendimiento necesarios.) Además, si el patrón de acceso cambia posteriormente a uno dominado por accesos a tablas individuales en lugar de juntas, debemos tener la posibilidad de cambiar nuevamente el esquema físico para que las tablas VP y P estén de nuevo físicamente separadas sin ningún efecto en el nivel lógico.

De lo anterior debe quedar claro que el problema de proporcionar independencia física de los datos es básicamente el problema de soportar la actualización de vistas (excepto que, como sucede con el problema de la actualización fragmentada que vimos en el capítulo 20, se manifiesta por sí mismo en un punto diferente de la arquitectura general del sistema). Ahora, en el capítulo 9 vimos que en teoría, *todas* las vistas relaciónales son actualizables. Por lo tanto, en teoría, si el esquema físico está derivado a partir del esquema lógico en la forma que describimos anteriormente, se logrará la máxima independencia física de los datos. Cualquier actualización expresada en términos del esquema lógico será traducible automáticamente en otra expresada en términos del esquema físico y *viceversa,* y los cambios al esquema físico no requerirán cambios al esquema lógico. *Nota:* De paso, comentamos que la *única* razón para hacer tales cambios al esquema físico deberá ser la de mejorar la eficiencia de almacenamiento o de rendimiento.

Sin embargo, desafortunadamente los productos SQL actuales no soportan adecuadamente la actualización de vistas. Por consecuencia, el conjunto de esquemas físicos permisibles está considerablemente (e innecesariamente) limitado en esos productos. Para ser más específicos, si (a) consideramos las tablas base en el nivel lógico como vistas, y las versiones guardadas de esas "vistas" en el nivel físico como tablas base, entonces (b) el esquema físico debe ser tal que el producto en cuestión pueda implementar todas las actualizaciones lógicamente posibles en esas "vistas" en términos de esas "tablas base". *Nota:* En la práctica tal vez sea posible simular el mecanismo de actualización de vistas adecuado por medio de procedimientos almacenados, procedimientos disparados, middleware o alguna combinación de ellos. Sin embargo, dichas técnicas están fuera del alcance del presente capítulo.

Diseño lógico

Las reglas del diseño lógico no dependen del uso que se pretenda dar a la base de datos, ya que se aplican las mismas reglas sin tomar en cuenta los tipos de aplicaciones. Por lo tanto, en particular no debe haber diferencia si esas aplicaciones son operacionales (OLTP) o de apoyo para la toma de decisiones; de cualquier forma, es necesario seguir el mismo procedimiento de diseño. Entonces, volvamos a ver las tres características *lógicas* de las bases de datos de apoyo para la toma de decisiones que identificamos casi al inicio de la sección 21.2 y consideremos sus implicaciones para el diseño lógico.

- *Combinaciones de columnas y muy pocas dependencias*

  Las consultas de apoyo para la toma de decisiones —y las actualizaciones, cuando son aplicables— con frecuencia tratan a las combinaciones de columnas como una unidad, lo que significa que las columnas componentes nunca son accedidas en forma individual (DIRECCIÓN es un ejemplo obvio). Acordemos referirnos a dicha combinación de columnas como una *columna compuesta*. Entonces, desde un punto de vista de diseño lógico, ¡tales columnas se comportan como si de hecho *no* fueran compuestas! Para ser más específicos, sea CC una columna compuesta y sea C alguna otra columna de la misma tabla. Entonces las dependencias que involucran a C y a los componentes de CC se reducen a dependencias que involucran a C y CC por sí mismas. Lo que es más, las dependencias que involucran a los componentes de *CC* y a ninguna otra columna, son irrelevantes y simplemente pueden ser ignoradas. El efecto neto es que la cantidad total de dependencias se reduce y el diseño lógico se vuelve más sencillo, con menos columnas y posiblemente hasta menos tablas.

  Nota: Sin embargo, es conveniente que mencionemos que el soporte completo y adecuado para las columnas compuestas no es trivial y se basa en el soporte de *tipos definidos por el usuario*. Para mayores explicaciones, vea la referencia [21.11] y los capítulos 5 y 25.

- *Las restricciones de integridad en general*

  Como ya explicamos, (a) las bases de datos de apoyo para la toma de decisiones son principalmente de sólo lectura y (b) la integridad de los datos se verifica al cargar (o actualizar) la base de datos y a menudo suponemos que no tiene sentido declarar restricciones de integridad en el esquema lógico. Sin embargo, éste no es el caso. Aunque es cierto que las restricciones nunca van a ser violadas (si la base de datos en realidad es de sólo lectura), no debemos desestimar el *valor semántico* de esas restricciones. Como vimos en el capítulo 8 (sección 8.10), las restricciones sirven para definir el significado de las tablas y el significado de toda la base de datos. Por lo tanto, declarar las restricciones proporciona un medio para decirle a los usuarios lo que significan los datos y por lo tanto, les ayuda en su tarea de formular consultas. Además, la declaración de restricciones también puede proporcionar información crucial al optimizador (vea la explicación de *optimization semántica* en el capítulo 17, sección 17.4).

  Nota: Declarar ciertas restricciones en ciertos productos SQL, ocasiona la creación automática de determinados índices y otros mecanismos de cumplimiento asociados, un hecho que puede incrementar significativamente el costo de las operaciones de carga y actualización. Este hecho, a su vez, puede servir para motivar a los diseñadores a evitar la declaración de restricciones. Sin embargo, el problema se deriva nuevamente de una confusión entre los asuntos lógicos y físicos; debe ser posible especificar restricciones de integridad en forma

declarativa al nivel *lógico* y expresar en forma independiente los mecanismos de cumplimiento correspondientes al nivel *físico*. Sin embargo, por desgracia los productos SQL actuales no diferencian adecuadamente entre los dos niveles (además, rara vez reconocen el I valor semántico de las restricciones). *Claves temporales*

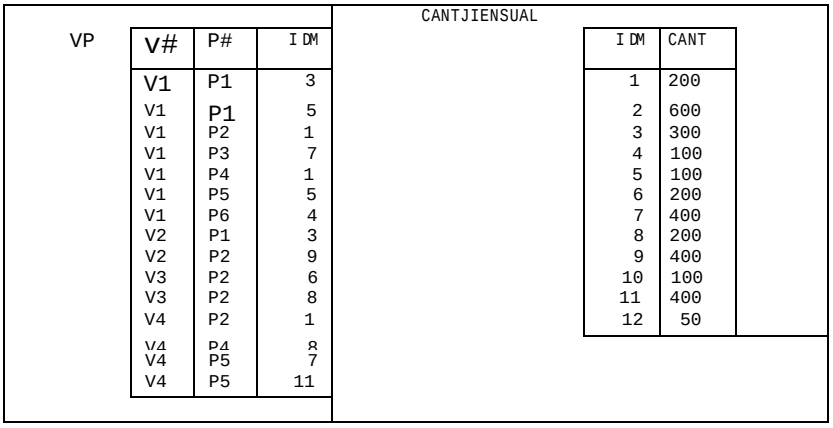
Por lo general, las bases de datos operacionales involucran sólo datos actuales. Por el con- j trario, las bases de datos de apoyo para la toma de decisiones involucran por lo general datos históricos y por lo tanto, tienden a poner *marcas de tiempo* en la mayoría o en todos los datos. Por consecuencia, las claves de dichas bases de datos incluyen frecuentemente columnas de marcas de tiempo. Por ejemplo, considere nuestra base de datos usual de proveedores y partes. Suponga que necesitamos extender la tabla para mostrar el mes particular (1 a 12) en el que sucedió cada envío. Entonces la tabla de envíos VP puede verse como muéstrala 1 figura 21.1. Observe que la columna adicional IDM ("ID de mes") es parte de la clave en I esta versión extendida de la tabla VP. Observe además que las consultas que involucran a VP ahora deben ser formuladas muy cuidadosamente para evitar el acceso a datos con marcas de tiempo diferentes (a menos que ese acceso sea exactamente lo que se desea, por supuesto). Tratamos brevemente estos temas en la sección 21.2; y el capítulo 22 los trataa profundidad.

Nota: Agregar columnas de marca de tiempo a una clave puede llevarnos a la necesidad de un diseño nuevo. Por ejemplo, suponga (algo artificialmente) que la cantidad de cada envío está determinada por el mes en el que sucede el envío (los datos de ejemplo de la figura 21.1 son consistentes con esta restricción). Luego, la versión modificada de la tabla VP satisface la dependencia funcional IDM —> CANT y por lo tanto, no está en la quinta —ni siquiera en la tercera— forma normal; por lo tanto, debe ser normalizada adicionalmente como lo indica la figura 21.2. Por desgracia, los diseñadores de apoyo para la toma de decisiones rara vez se preocupan por tomar en cuenta tales *dependencias inducidas.*

VP

V#	P#	IDM	CANT
V1	P1	3	300
V1	P1	5	100
V1	P2	1	200
V1	P3	7	400
V1	P4	1	200
V1	P5	5	100
V1	P6	4	100
V2	P1	3	300
V2	P2	9	400
V3	P2	6	200
V3	P2	8	200
V4	P2	1	200
V4	P4	8	200
V4	P5	7	400
V4	P5	11	400

Figura 21.1 Valores de ejemplo para la tabla VP incluyendo IDs de mes.



VP

V#	P#	IDM
V1	P1	3
V1	P1	5
V1	P2	1
V1	P3	7
V1	P4	1
V1	P5	5
V1	P6	4
V2	P1	3
V2	P2	9
V3	P2	6
V3	P2	8
V4	P2	1
V4	P4	8
V4	P5	7
V4	P5	11

CANTJIENSUAL

IDM	CANT
1	200
2	600
3	300
4	100
5	100
6	200
7	400
8	200
9	400
10	100
11	400
12	50

Figura **21.2** Contraparte normalizada de la figura 21.1.

Diseño físico

En la sección 21.2 dijimos que las bases de datos de apoyo para la toma de decisiones tienden a ser *grandes y fuertemente indexadas,* e involucran diversos tipos de *redundancia controlada.* En esta subsección trataremos brevemente estas cuestiones de diseño físico.

Primero consideramos el **partido** (también conocido como *fragmentación).* El partido representa un ataque al problema de la "base de datos grande"; divide una tabla dada en un conjunto de *particiones o fragmentos* separados para efectos de almacenamiento físico (vea la explicación sobre fragmentación en el capítulo 20). Dichas particiones pueden mejorar significativamente el manejo y la accesibilidad de la tabla en cuestión. Por lo general, a cada partición se le asignan determinados recursos de hardware más o menos específicos (por ejemplo, disco, CPU) y por lo tanto, se minimiza la competencia por dichos recursos entre las particiones. Las tablas son partidas horizontalmente* por medio de *una. función de partición,* la cual toma valores de columnas seleccionadas (la *clave de partición)* como argumentos y regresa un número o dirección de partición. Por lo general, dichas funciones soportan la partición por rango, por dispersión y en ronda, entre otros tipos (vea el comentario a la referencia [17.58] en el capítulo 17).

Ahora pasemos al **indexado.** Por supuesto, es bien conocido que el uso del tipo adecuado de índice puede reducir drásticamente la E/S. La mayoría de los primeros productos SQL proporcionaban solamente un tipo de índice, el árbol B, pero a través de los años se han tenido otros tipos disponibles, en especial en conexión con las bases de datos de apoyo para la toma de decisiones; éstos incluyen a los índices de *mapa de bits, dispersión, multitabla, lógicos y funcionales,* así como a los índices de árbol B en sí. Comentaremos brevemente cada uno de estos tipos.

- *índices de árbol B.* Los índices de árbol B proporcionan acceso eficiente para consultas de alcance (a menos que la cantidad de filas accedidas llegue a ser demasiado grande). La actualización de árboles B es relativamente eficiente.

---

*E1 partido vertical, aunque pudiera ser ventajoso, no es muy usado, ya que la mayoría de los productos no lo soporta.

- *índices de mapa de bits.* Supongamos que la tabla indexada T contiene *n* filas. Entonces un índice de mapa de bits sobre la columna C de la tabla T, guarda un vector de *n* bits para cada valor que está en el dominio de C, y enciende el bit correspondiente a la fila R cuando esa fila contiene el valor aplicable en la columna C. Estos índices son eficientes para las consultas que involucran conjuntos de valores, aunque llegan a ser menos eficientes cuando los conjuntos se vuelven demasiado grandes. Observe en particular que varias operaciones relacionales (juntas, uniones, restricciones de igualdad, etcétera) pueden ser realizadas completamente dentro de los índices, por medio de operaciones lógicas simples (AND, OR, NOT) sobre los vectores de bits; el acceso a los datos actuales no es en absoluto necesario sino hasta que se tiene que recuperar el resultado final. La actualización de índices de mapa de bits es relativamente ineficiente.
- *índices de dispersión* (también conocidos como *direccionamiento de dispersión* o simplemente *dispersión [hashing]).* Los índices de dispersión son eficientes para acceder a filas específicas (no rangos). El costo de la computación es lineal para la cantidad de filas, siempre y cuando la función de dispersión no necesite ser extendida para acomodar valores de clave adicionales. La dispersión también puede ser usada para implementar juntas de manera eficiente, tal como lo describimos en el capítulo 17.
- *índices multitabla* (también conocidos como *índices de junta).* Una entrada de índice multitabla contiene esencialmente apuntadores hacia filas de varias tablas, en lugar de sólo hacia filas de una tabla. Dichos índices pueden mejorar el rendimiento de las juntas y el proceso de verificación de las restricciones de integridad de las multitablas (es decir, de la base de datos).
- *índices lógicos* (por lo general, mejor conocidos como índices de *expresión).* Un índice lógico indica para qué filas de una tabla específica, una expresión lógica específica (que involucra columnas de la tabla en cuestión) da como resultado *verdadero.* Dichos índices son particularmente valiosos cuando la expresión lógica relevante es un componente común de las condiciones de restricción.
- *índices funcionales.* Un índice funcional indexa las filas de una tabla no con base en los valores de esas filas, sino con base en el resultado del llamado a alguna función especificada sobre esos valores.

Además de todo lo anterior, han sido propuestos varios tipos de índices *híbridos* (combinaciones de los anteriores). El valor de dichos híbridos es difícil de caracterizar en términos generales. También se han propuesto una enorme cantidad de tipos de índices *especializados* (por ejemplo, *árboles R,* que están pensados para manejar datos geométricos). En este libro, no intentaremos realizar la enorme tarea de describir todos estos tipos de índices; para una explicación amplia, vea por ejemplo la referencia [25.27].

Por último, pasemos al asunto de la **redundancia controlada.** La redundancia controlada es una herramienta importante para reducir la E/S y minimizar la contienda. Como explicamos en el capítulo 1, la redundancia está controlada cuando es administrada por el DBMS y está oculta para los usuarios. (Observe que, por definición, la redundancia que es controlada adecuadamente en el nivel físico es invisible en el nivel lógico, y por lo tanto, no tiene efecto sobre la corrección de ese nivel lógico.) Hay dos tipos amplios de esta redundancia:

- El primero implica mantener copias exactas, o *réplicas,* de los datos básicos. *Nota:* Lo que puede ser considerado como una forma de replicación menos ambiciosa, la *administración de copias,* también es ampliamente soportada (vea más adelante).

- El segundo implica mantener *datos derivados* además de los datos básicos, muy frecuente mente en la forma de *tablas de resumen* o de *columnas calculadas* o *derivadas.*

Trataremos cada uno de éstos por separado.

En el capítulo 20 (secciones 20.3 y 20.4) explicamos los conceptos básicos de la **replicación** (vea especialmente la subsección "Propagación de la actualización" en la sección 20.4); aquí simplemente repetiremos algunos puntos sobresalientes de esas explicaciones y haremos algunos comentarios adicionales. Recuerde primero que la replicación puede ser *síncrona* o *asincrona:*

- En el caso *síncrono,* si se actualiza una réplica dada, todas las demás réplicas del mismo frag mento de datos también se actualizan dentro de la misma transacción; lo que implica que (desde un punto de vista lógico) sólo existe una versión de los datos. La mayoría de los pro ductos implementan la replicación síncrona por medio de procedimientos disparados (posi blemente ocultos y manejados por el sistema). Sin embargo, la replicación síncrona tiene la desventaja de que impone una sobrecarga sobre todas las transacciones que actualizan cual quier réplica (también puede haber problemas de disponibilidad).
- En el caso *asincrono,* las actualizaciones a una réplica son propagadas hacia las demás en algún momento posterior, *no* dentro de la misma transacción. Por lo tanto, la replicación asin crona presenta un *retardo de tiempo,* o *latencia,* durante el cual es posible que las réplicas no sean idénticas (y por lo tanto, el término "réplica" ya no es muy adecuado, debido a que ya no estamos hablando de copias exactas). La mayoría de los productos implementan la replicación asincrona leyendo la bitácora de transacciones o una cola estable de actualiza ciones que necesiten ser propagadas.

  La ventaja de la replicación asincrona es que la sobrecarga de replicación está desacoplada con relación a la transacción de actualización, la cual puede ser de "misión crítica" y altamente sensible al rendimiento. La desventaja es que los datos pueden llegar a ser inconsistentes (en cuanto a como son vistos por el usuario); es decir, la redundancia puede ser mostrada a través del nivel lógico, lo que significa en términos estrictos que el término "redundancia controlada" ya no es muy adecuado.*

Hacemos notar que al menos en el mundo comercial, el término "replicación" ha llegado a significar principalmente (de hecho, casi exclusivamente) replicación *asincrona* en particular (como dijimos en el capítulo 20).

La diferencia básica entre replicación y **administración de copias** es la siguiente. Con la replicación, las actualizaciones para una réplica son (tarde o temprano) propagadas "automáticamente" hacia todas las demás. Por el contrario, con la administración de copias no hay tal propagación automática; en su lugar, las copias de datos son creadas y mantenidas por medio de algún proceso por lotes —o en segundo plano— que está desacoplado en el tiempo con respecto a las transacciones de actualización. La administración de copias es generalmente más eficiente que la replicación, ya

---

*Observe también que las réplicas pueden llegar a ser inconsistentes en formas que son difíciles de evitar y de componer. En particular, es posible presentar conflictos sobre el orden en que las transacciones son aplicadas. Por ejemplo, dejemos que la transacción TI inserte una fila en la réplica RX, que la transacción T2 luego borre esa fila y que RY sea una réplica de RX. Si las actualizaciones son propagadas hacia RY, pero llegan a RY en orden inverso (por ejemplo, a causa de retardos en las rutas), T2 encuentra que no hay fila que borrar y luego TI inserta la fila. El efecto real es que RY contiene la fila y en cambio, RX no la tiene. La administración de conflictos y el cumplimiento de la consistencia a través de réplicas son problemas difíciles que están fuera del alcance de este libro.

que es posible copiar grandes cantidades de datos de una sola vez. La desventaja es que la mayor parte del tiempo las copias no son idénticas a los datos básicos; además, los usuarios deben (por lo general) estar conscientes de en qué momento han sido sincronizadas. Por lo general, la administración de copias se simplifica al requerir que las actualizaciones sean aplicadas de acuerdo con algún tipo de esquema de "copia primaria" (vea el capítulo 20).

El otro tipo de redundancia que aquí consideramos son las **columnas calculadas y** las **tablas de resumen.** Estas construcciones son particularmente importantes en el contexto del apoyo a la toma de decisiones, ya que se usan para guardar valores de datos precalculados (es decir, valores que son calculados a partir de otros datos guardados en algún otro lugar de la base de datos). Para ser más específicos, dichas construcciones evitan la necesidad de volver a calcular dichos valores cada vez que sean requeridos en alguna consulta. Una *columna calculada* es aquella cuyo valor en cualquier fila dada se deriva (de cierta forma) a partir de otros valores de la misma fila.* Una *tabla de resumen* es una tabla que guarda totales (sumas, promedios, cuentas, etcétera) de valores que están en otras tablas. Con frecuencia, dichos totales son precalculados para varios agrupamientos diferentes de los mismos datos de detalle (vea la sección 21.6). *Nota:* Si las columnas calculadas y las tablas de resumen van a ser en verdad ejemplares de redundancia controlada, entonces deben estar completamente ocultas ante los usuarios; sin embargo, en los productos actuales usualmente no es así.

Las tablas de resumen y las columnas calculadas son implementadas casi siempre por medio de procedimientos disparados que son administrados por el sistema, aunque también pueden ser implementadas por medio de código de procedimientos escrito por el usuario. El primer enfoque permite mantener la consistencia entre los datos básicos y los derivados (siempre y cuando ambos sean actualizados en la misma transacción, lo cual puede o no ser el caso; incluso si lo es, observe que un alto nivel de aislamiento puede ser crucial para obtener esa consistencia). Es más probable que el segundo enfoque exponga inconsistencia ante el usuario.

Errores comunes de diseño

En esta subsección comentaremos brevemente algunas prácticas de diseño que son comunes en el ambiente de apoyo para la toma de decisiones y que seguimos considerando poco convenientes;

- *Filas duplicadas.* Los diseñadores de apoyo para la toma de decisiones dicen con frecuencia que sus datos simplemente no tienen un identificador único y que por lo tanto, tienen que permitir duplicados. Las referencias [5.3] y [5.6] explican en detalle por qué los duplicados son un error; aquí simplemente comentamos que el "requerimiento" surge debido a que el esquema físico no se deriva a partir de un esquema lógico (el cual probablemente nunca se creó). Observamos también que en tales diseños, las filas tienen a menudo significados no uniformes (en especial si está presente cualquier nulo); es decir, no todas son ejemplares del mismo predicado (vea el capítulo 3, sección 3.4, y también el capítulo 18). *Nota:* En ocasiones, los duplicados se permiten incluso deliberadamente, en especial cuando el diseñador tiene una formación orientada a objetos (vea el último párrafo de la sección 24.2 en el capítulo 24).

---

*En forma alterna, el valor calculado podría ser derivado a partir de los valores de varias filas de la misma tabla o de otras tablas. Sin embargo, dicho enfoque implica que la actualización de una fila podría requerir que también se actualizaran muchas otras filas; en particular, esto puede tener un efecto muy negativo sobre las operaciones de carga y actualización. /

Desnormalización y prácticas relacionadas. En un esfuerzo erróneo para eliminar juntas y reducir la E/S, a menudo los diseñadores prejuntan tablas, introducen columnas derivadas de diversos tipos, etcétera. Dichas prácticas pueden ser aceptables en el nivel físico, pero no si son detectables en el nivel lógico.

Esquemas de estrella. Con mucha frecuencia, los "esquemas de estrella" (también conocidos como esquemas *dimensionales)* son el resultado de intentar "tomar atajos" en una técnica adecuada de diseño. Es poco lo que se puede ganar con esos atajos. Con frecuencia afectan el rendimiento y la flexibilidad conforme crece la base de datos, y la resolución de tales dificultades por medio del rediseño físico fuerza también a hacer cambios en las aplicaciones (ya que en realidad los esquemas de estrella son esquemas *físicos,* aunque estén expuestos a las aplicaciones). El problema general yace en la naturaleza *ad hoc* del diseño. *Nota:* Trataremos los esquemas de estrella con mayor detalle en la sección 21.5. *Nulos.* Los diseñadores tratan frecuentemente de ahorrar espacio permitiendo nulos en las columnas (este truco *puede* funcionar si la columna en cuestión es de algún tipo de dato de longitud variable y el producto en cuestión representa a los nulos en dichas columnas por medio de cadenas vacías en el nivel físico). Sin embargo, por lo general dichos intentos son erróneos. No sólo es posible (y necesario) hacer un diseño que —en primer lugar— evite los nulos [18.20], sino que los esquemas resultantes proporcionan a menudo una mejor eficiencia de almacenamiento y un mejor rendimiento de E/S.

Diseño de tablas de resumen. La cuestión del diseño lógico de tablas de resumen es con frecuencia ignorada, lo que da lugar a una redundancia no controlada y a dificultades para mantener la consistencia. Como consecuencia, los usuarios pueden llegar a confundirse con respecto al significado de los datos de resumen y a la manera de formular consultas que los involucren. Para evitar tales problemas, todas las tablas de resumen en el mismo nivel de agregación (o sea, de totales; vea la sección 21.6) deben ser diseñadas como si formaran una base de datos por derecho propio. Luego podemos evitar determinados problemas de *actualización cíclica* (a) prohibiendo que las actualizaciones abarquen varios niveles de agregación y (b) sincronizando las tablas de resumen al hacer siempre las agregaciones (totalizar) del nivel de detalle hacia arriba.

"Varias rutas de navegación". A menudo, los diseñadores de apoyo para la toma de decisiones y los usuarios dicen (incorrectamente) que hay una "multiplicidad de rutas de navegación" hacia algún dato deseado, cuando en realidad quieren decir que los mismos datos pueden ser alcanzados por medio de varias expresiones relaciónales diferentes. A veces las expresiones en cuestión en realidad son equivalentes, como en el caso de —por ejemplo— A JOIN (B JOIN C) y (A JOIN B) JOIN C (vea el capítulo 17); en ocasiones son equivalentes sólo debido a que hay alguna restricción de integridad que hace que en efecto sea así (de nuevo vea el capítulo 17); en ocasiones, ¡no son equivalentes en absoluto! Como ejemplo de esto último, suponga que las tablas A, B y C tienen una columna común K; entonces, "seguir la ruta de K desde A hacia B y por lo tanto, hacia C" en realidad *no* es (generalmente) lo mismo que "seguir la ruta de K directamente desde A hacia C".

Es claro que los usuarios pueden llegar a confundirse en tales casos y no estar seguros de qué expresión usar o de si habrá alguna diferencia o no en el resultado. Por supuesto, parte de este problema sólo puede ser resuelta mediante una enseñanza adecuada para el usuario. Otra parte puede ser resuelta si el optimizador hace su trabajo adecuadamente. Sin embargo, otra parte se debe a que los diseñadores permiten redundancias en el esquema lógico o permiten que los usuarios accedan directamente al esquema físico, y *esa* parte del problema sólo puede ser resuelta mediante una práctica de diseño adecuada.

En resumen, creemos que muchas de las dificultades de diseño que supuestamente se presentan por los requerimientos para el apoyo a la toma de decisiones, pueden ser resueltas siguiendo un enfoque disciplinado. Además, muchas de estas dificultades son *causadas* por no seguir este enfoque (aunque es conveniente añadir que a veces se agravan por problemas con SQL).

21.4 PREPARACIÓN DE LOS DATOS

Muchas de las cuestiones que rodean a los sistemas de apoyo para la toma de decisiones, se refieren en primer lugar a las tareas de obtener y preparar los datos. Los datos deben ser *extraídos* de diversas fuentes, *limpiados, transformados* y *consolidados, cargados* en la base de datos de apoyo para la toma de decisiones y luego *actualizados* periódicamente. Cada una de estas operaciones involucra sus propias consideraciones especiales.* Las examinamos una por una y luego concluimos la sección con una breve explicación de los *almacenes de datos operacionales.*

Extracción

La **extracción** es el proceso de capturar datos de las bases de datos operacionales y otras fuentes. Hay muchas herramientas disponibles para ayudar en esta tarea, incluyendo herramientas proporcionadas por el sistema, programas de extracción personalizados y productos de extracción comerciales (de propósito general). El proceso de extracción tiende a ser intensivo en E/S y por lo tanto, puede interferir con las operaciones de misión crucial; por esta razón, este proceso a menudo es realizado en paralelo (es decir, como un conjunto de subprocesos paralelos) y en un nivel físico. Sin embargo, dichas "extracciones físicas" pueden ocasionar problemas para el procesamiento subsecuente, ya que pueden perder información —en especial información de vínculos— que está representada de alguna manera física (por ejemplo, por apuntadores o por contigüidad física). Por esta razón, los programas de extracción proporcionan en ocasiones un medio para preservar dicha información introduciendo números de registro secuenciales y reemplazando apuntadores por lo que en realidad son valores de clave externa.

Limpieza

Pocas fuentes de datos controlan adecuadamente la calidad de los datos. Por consecuencia, los datos requieren frecuentemente de una **limpieza** (por lo general, por lote) antes de que puedan ser introducidos en la base de datos de apoyo para la toma de decisiones. Las operaciones de limpieza típicas incluyen el llenado de valores faltantes, la corrección de errores tipográficos y otros de captura de datos, el establecimiento de abreviaturas y formatos estándares, el reemplazo de sinónimos por identificadores estándares, etcétera. Los datos que son erróneos y que no pueden ser limpiados, serán reemplazados. *Nota:* En ocasiones, la información obtenida durante el proceso de limpieza puede ser usada para identificar la causa de los errores en el origen y por lo tanto, mejorar la calidad de los datos a través del tiempo.

---

*De paso, comentamos que estas operaciones a menudo pueden beneficiarse de las posibilidades en el nivel de conjunto de los sistemas relaciónales, aunque en la práctica rara vez lo hacen.

Transformación y consolidación

Aun después haber sido limpiados, es probable que los datos todavía no estén en la forma en que se requieren para el sistema de apoyo para la toma de decisiones y por lo tanto, deberán ser **transformados** adecuadamente. Por lo general, la forma requerida será un conjunto de archivos, uno por cada tabla identificada en el esquema físico; como resultado, la transformación de los datos puede involucrar la división o la combinación de registros fuente de acuerdo a lo que hemos explicado en el capítulo 1 (sección 1.5). *Nota:* A veces, los errores de datos que no fueron corregidos durante la limpieza son encontrados durante el proceso de transformación. Como dijimos antes, por lo general cualquier dato incorrecto es rechazado. (También, igual que antes, la información obtenida como parte de este proceso puede ser usada, en ocasiones, para mejorar la calidad de la fuente de datos.)

La transformación es particularmente importante cuando necesitan mezclarse varias fuentes de datos, un proceso al que se llama **consolidación.** En estos casos, cualquier vínculo implícito entre datos de distintas fuentes necesita volverse explícito (introduciendo valores de datos explícitos). Además, las fechas y horas asociadas con el significado que tienen los datos en los negocios, necesitan ser mantenidas y correlacionadas entre fuentes; un proceso llamado "sincronización en el tiempo" [¡cita textual!].

Por razones de rendimiento, las operaciones de transformación se realizan frecuentemente en paralelo. Pueden ser intensivas tanto en E/S como en CPU.

Nota: La sincronización en el tiempo puede ser un problema difícil. Por ejemplo, suponga que queremos encontrar el promedio de las ganancias por cliente y por vendedor en cada trimestre. Suponga que los datos del cliente contra las ganancias son mantenidos por trimestre fiscal en una base de datos de contabilidad, y en cambio, los datos del vendedor contra el cliente son mantenidos por trimestre de calendario en una base de datos de ventas. De manera clara, necesitamos fusionar los datos de las dos bases de datos. La consolidación de clientes es fácil, involucra simplemente la coincidencia de IDs de clientes. Sin embargo, la cuestión de la sincronización de tiempo es mucho más difícil. Podemos encontrar las ganancias de cliente por trimestre *fiscal* (a partir de la base de datos de contabilidad), pero no podemos decir qué vendedores fueron responsables de cuáles clientes en ese momento y, a fin de cuentas, no podemos encontrar las ganancias de clientes por trimestre de *calendario.*

Carga

Los fabricantes de DBMS han puesto considerable importancia en la eficiencia de las operaciones de **carga.** Para los propósitos actuales, consideramos que las "operaciones de carga" incluyen (a) el movimiento de los datos transformados y consolidados hacia la base de datos de apoyo para la toma de decisiones, (b) la verificación de su consistencia (es decir, verificación de integridad) y (c) la construcción de cualquier índice necesario. Comentaremos brevemente cada paso:

a. *Movimiento de datos.* Por lo general, los sistemas modernos proporcionan herramientas de car ga en paralelo. En ocasiones formatearán previamente los datos para darles el formato físico interno requerido por el DBMS de destino antes de la carga real. (Una técnica alterna que pro porciona gran parte de la eficiencia de las cargas preformateadas es cargar los datos en tablas de trabajo que se asemejan al esquema de destino. La verificación de la integridad necesaria puede ser realizada en esas tablas de trabajo [vea el párrafo b] y luego usar los INSERTS en el nivel de conjunto para mover los datos desde las tablas de trabajo hacia las tablas de destino.)

b. *Verificación de integridad.* La mayor parte de la verificación de integridad de los datos a ser cargados puede ser realizada antes de la carga real, sin hacer referencia a los datos que

ya están en la base de datos. Sin embargo, ciertas restricciones no pueden verificarse sin examinar la base de datos existente; por ejemplo, una restricción de unicidad tendrá que ser verificada, por lo general, durante la carga real (o por lotes después de que se haya terminado la carga).

c. *Construcción de índices.* La presencia de índices puede hacer significativamente lento el proceso de carga, debido a que la mayoría de los productos actualiza los índices conforme cada fila es insertada en la tabla subyacente. Por esta razón, en ocasiones es buena idea eliminar los índices antes de la carga y luego volverlos a crear. Sin embargo, este enfoque no vale la pena cuando la proporción de los nuevos datos (con respecto a los existentes) es pequeña, ya que el costo de crear un índice no se compensa con el tamaño de la tabla a indexar. Además, la creación de un índice grande puede estar sujeta a errores de asignación irrecuperables, y entre más grande sea el índice, es más probable que ocurran tales errores. *Nota:* La mayoría de los productos DMBS soportan la creación de índices en paralelo en un esfuerzo para agilizar los procesos de carga y de construcción de índices.

Actualización

La mayoría de las bases de datos de apoyo para la toma de decisiones (aunque no todas) requieren una **actualización** periódica de los datos para mantenerlos razonablemente vigentes. La actualización involucra por lo general una carga parcial, aunque algunas aplicaciones de apoyo para la toma de decisiones requieren la eliminación de lo que hay en la base de datos y una recarga completa. La actualización involucra todos los problemas que están asociados con la carga, pero también es probable que deba realizarse mientras los usuarios están accediendo a la base de datos. Vea el capítulo 9, sección 9.5, y también las referencias [9.2] y [9.6].

Almacenes de datos operacionales

Un ODS **(almacén de datos operacionales)** es una "colección de datos actuales —o casi actuales— integrados y volátiles (es decir actualizables)" que están orientados a un tema [21.19]. En otras palabras, es un tipo especial de base de datos. El término *orientado a un tema* significa que los datos en cuestión tienen que ver con alguna área temática específica (por ejemplo, clientes, productos, etcétera). Un almacén de datos operacionales puede ser usado (a) como un área transitoria para la reorganización física de los datos operacionales extraídos, (b) para proporcionar informes operacionales y (c) para apoyar la toma de decisiones operacionales. También puede servir (d) como un punto de consolidación si es que los datos operacionales proceden de varias fuentes. Por lo tanto, los ODSs sirven para muchos propósitos. *Nota:* Debido a que no acumulan datos históricos (por lo general), no crecen demasiado; por otro lado, están generalmente sujetos a una actualización muy frecuente, o incluso continua, a partir de fuentes de datos operacionales.* Los problemas de sincronización en el tiempo (vea la subsección anterior "transformación y consolidación") pueden ser atacados satisfactoriamente dentro de un ODS si la actualización es suficientemente frecuente.

---

*A veces, para este propósito se usa la replicación asincrona desde las fuentes de datos operacionales hacia el ODS. De esta forma, los datos con frecuencia pueden actualizarse en cuestión de minutos.

21.5 DATA WAREHOUSES Y DATA MARTS

Por lo general, los sistemas operacionales tienen requerimientos de rendimiento estrictos, cargas de trabajo predecibles, pequeñas unidades de trabajo y una alta utilización. Por el contrario, los sistemas de apoyo para la toma de decisiones tienen por lo general requerimientos de rendimiento variantes, cargas de trabajo impredecibles, grandes unidades de trabajo y utilización errática. Estas diferencias pueden hacer que sea muy difícil combinar el procesamiento operacional y el de apoyo para la toma de decisiones dentro de un solo sistema, en especial con relación a la planeación de la capacidad, la administración de recursos y el perfeccionamiento del rendimiento del sistema. Por estas razones, en general los administradores de sistemas operacionales están poco dispuestos a permitir actividades de apoyo para la toma de decisiones en sus sistemas; ésta es la causa del familiar enfoque del sistema doble.

Nota: Comentamos adicionalmente que esto no siempre fue así; los primeros sistemas de apoyo para la toma de decisiones fueron ejecutados en los sistemas operacionales pero con una baja prioridad, o durante la llamada "ventana por lotes". Con recursos de computación suficientes, existen varias ventajas de este arreglo, y tal vez la más obvia es que evita todas las operaciones (posiblemente costosas) de copiado de datos, reformateo y transferencia (etcétera) requeridas por el enfoque del sistema doble. De hecho, el valor de la integración de las actividades operacionales y de apoyo para la toma de decisiones está llegando a ser cada vez más reconocido. Sin embargo, más detalles de tal integración están fuera del alcance de este capítulo.

A pesar del párrafo anterior (al menos al momento de la publicación de este libro), permanece el hecho de que los datos de apoyo para la toma de decisiones necesitan por lo general ser recolectados a partir de una variedad de sistemas operacionales (a menudo dispares) y ser mantenidos en un almacén de datos propio en una plataforma independiente. Ese almacén de datos separados es un *data warehouse.*

Data warehouses

Al igual que los almacenes de datos operacionales (y los data marts, vea la siguiente subsección), un **data warehouse** es un tipo especial de base de datos. Al parecer, el término se originó a finales de los ochenta [21.13], [21.17], aunque el concepto es de alguna manera más antiguo. La referencia [21.18] define un data warehouse como "un almacén de datos orientado a un tema, integrado, no volátil y variante en el tiempo, que soporta decisiones de administración" (donde el término *no volátil* significa que una vez que los datos han sido insertados, no pueden ser cambiados, aunque sí pueden ser borrados). Los data warehouses surgieron por dos razones: primero, la necesidad de proporcionar una fuente única de datos limpia y consistente para propósitos de apoyo para la toma de decisiones; segundo, la necesidad de hacerlo sin afectar a los sistemas operacionales.

Por definición, las cargas de trabajo del data warehouse están destinadas para el apoyo a la toma de decisiones y por lo tanto, tienen consultas intensivas (con actividades ocasionales de inserción por lotes); asimismo, los propios data warehouses tienden a ser bastante grandes (a menudo mayores que 500GB y con una tasa de crecimiento de hasta el 50 por ciento anual). Por consecuencia, es difícil —aunque no imposible— perfeccionar el rendimiento. También puede ser un problema la escalabilidad. Contribuyen a ese problema (a) los errores de diseño de la base de datos (que tratamos en la subsección final de la sección 21.3), (b) el uso ineficiente de los operadores relaciónales (que mencionamos brevemente en la sección 21.2), (c) la debilidad en la implementación del modelo relacional del DBMS, (d) la falta de escalabilidad del propio DBMS y (e) los errores de diseño

arquitectónico que limitan la capacidad e imposibilitan la escalabilidad de la plataforma. Los puntos (d) y (e) están fuera del alcance de este libro; por el contrario, ya hemos explicado los puntos (a) y (b) en este capítulo, y el punto (c) se trata ampliamente en otras partes del libro.

Data marts

Por lo general, los data warehouses están hechos para proporcionar una fuente de datos única para todas las actividades de apoyo para la toma de decisiones. Sin embargo, cuando los data warehouses se hicieron populares (a principio de los años noventa) pronto fue evidente que los usuarios a menudo realizaban amplias operaciones de informes y análisis de datos sobre un subconjunto relativamente pequeño de todo el data warehouse. Asimismo, era muy probable que los usuarios repitieran las mismas operaciones sobre el mismo subconjunto de datos cada vez que era actualizado. Además, algunas de esas actividades —por ejemplo, análisis de pronósticos, simulación, modelado de datos de negocios del tipo "qué pasaría si..."— involucraban la creación de nuevos esquemas y datos con actualizaciones posteriores a esos nuevos datos.

De manera obvia, la ejecución repetida de tales operaciones sobre el mismo subconjunto de todo el almacén no era muy eficiente; por lo tanto, pareció obviamente una buena idea construir algún tipo de "almacén" limitado de propósito general que estuviera hecho a la medida de ese propósito. Además, en algunos casos sería posible extraer y preparar los datos requeridos directamente a partir de las fuentes locales, lo que proporcionaba un acceso más rápido a los datos que si tuvieran que ser sincronizados con los demás datos cargados en todo el data warehouse. Dichas consideraciones condujeron al concepto de **data marts.**

De hecho, hay alguna controversia sobre la definición precisa del término *data mart.* Para nuestros propósitos podemos definirlo como "un almacén de datos especializado, orientado aun tema, integrado, volátil y variante en el tiempo para apoyar un subconjunto específico de decisiones de administración". Como puede ver, la principal diferencia entre un data mart y un data warehouse es que el data mart es *especializado* y *volátil.* Por *especializado* queremos decir que contiene datos para dar apoyo (solamente) a un área específica de análisis de negocios; por *volátil* queremos decir que los usuarios pueden actualizar los datos e incluso, posiblemente, crear nuevos datos (es decir, nuevas tablas) para algún propósito.

Hay tres enfoques principales para la creación de un data mart:

- Los datos pueden ser simplemente extraídos del data warehouse; de hecho, sigue un enfo que de "divide y vencerás" sobre la carga de trabajo general de apoyo para la toma de decisio nes, a fin de lograr un mejor rendimiento y escalabilidad. Por lo general, los datos extraídos son cargados en una base de datos que tiene un esquema físico que se parece mucho al sub conjunto aplicable del data warehouse; sin embargo, puede ser simplificado de alguna mane ra gracias a la naturaleza especializada del data mart.
- A pesar del hecho de que el data warehouse pretende proporcionar un "punto de control úni co", un data mart puede ser creado todavía en forma independiente (es decir, *no* por medio de la extracción a partir del data warehouse). Dicho enfoque puede ser adecuado si el data ware house es inaccesible por alguna razón, digamos razones financieras, operacionales o incluso políticas (o puede ser que ni siquiera exista todavía el data warehouse; vea el siguiente punto).
- Algunas instalaciones han seguido un enfoque de "primero el data mart", donde los data marts son creados conforme van siendo necesarios y el data warehouse general es creado, finalmente, como una consolidación de los diversos data marts.

Los últimos dos enfoques sufren posibles problemas de desacoplo semántico. Los data marts independientes son particularmente susceptibles a tales problemas, debido a que no hay forma obvia de verificar los desacoplos semánticos cuando las bases de datos son diseñadas en forma independiente. Por lo general, la consolidación de data marts en data warehouses falla, a menos que (a) se construya primero un esquema lógico único para el data warehouse y (b) los esquemas para los data marts individuales se deriven después a partir del esquema del data warehouse. (Por supuesto, el esquema de este último punto puede evolucionar —suponiendo que se sigan buenas prácticas de diseño— para incluir el tema de cada nuevo data mart conforme sea necesario.)

Una nota sobre el diseño de data marts: Una decisión importante que hay que tomar en el diseño de cualquier base de datos de apoyo para la toma de decisiones es la **granularidad** de la misma. Aquí el término *granularidad* se refiere al nivel más bajo de agregación de datos que se mantendrá en la base de datos. Ahora bien, la mayona de las aplicaciones de apoyo para la toma de decisiones requerirán tarde o temprano acceso a datos detallados y por lo tanto, la decisión será fácil para el data warehouse. Para un data mart puede ser más difícil. La extracción de grandes cantidades de datos detallados del data warehouse, y su almacenamiento en el data mart, puede ser muy ineficiente si ese nivel de detalle no se necesita con mucha frecuencia. Por otro lado, en algunas ocasiones es difícil establecer definitivamente cuál es el nivel más bajo de agregación que en realidad se necesita. En dichos casos, los datos detallados pueden ser accedidos directamente desde el data warehouse cuando se necesiten, manteniendo en el data mart los datos que de alguna manera ya fueron agregados. Al mismo tiempo, generalmente no se hace la agregación completa de los datos, debido a que las diversas formas en las que pueden ser agregados, generarían cantidades muy grandes de datos de resumen. Trataremos este punto con mayor detalle en la sección 21.6.

Un punto adicional: Debido a que los usuarios de los data marts emplean frecuentemente determinadas herramientas analíticas, el diseño físico a menudo está determinado, en parte, por las herramientas específicas a usar (vea la explicación de "ROLAP contra MOLAP" en la sección 21.6). Esta circunstancia desafortunada puede conducir a "esquemas dimensionales" (tratados a continuación) que no son soportados por las buenas prácticas de diseño relacional.

Esquemas dimensionales

Suponga que queremos recolectar un historial de las transacciones de negocios para efectos de análisis. Como indicamos en la sección 21.1, los primeros sistemas de apoyo para la toma de decisiones mantenían generalmente ese historial como un archivo simple, el cual podía ser accedido más tarde por medio de una exploración secuencial. Sin embargo, conforme el volumen de los datos se incrementa, llega a ser cada vez más necesario apoyar la búsqueda con el acceso directo a ese archivo desde varias perspectivas diferentes. Por ejemplo, tener la posibilidad de encontrar todas las transacciones de negocios que involucran a un producto en particular o todas las transacciones de negocios que suceden dentro de un periodo en particular o todas las transacciones de negocios que se refieren a un cliente en particular.

Un método de organización que soporta este tipo de acceso fue llamado base de datos "multicatálogo".* Para continuar con nuestro ejemplo, dicha base de datos consistiría en un gran archivo central de datos que contuviera los datos de las transacciones de negocios, junto con tres archivos de "catálogo" individuales para productos, periodos y clientes, respectivamente. Dichos

*No tiene nada que ver con los catálogos en el sentido que tiene el término en las bases de datos modernas.

archivos de catálogo se parecen a los índices dado que contienen apuntadores hacia los registros que están en el archivo de datos; sin embargo, (a) las entradas pueden ser colocadas en los archivos explícitamente por el usuario ("mantenimiento del catálogo") y (b) los archivos pueden contener información suplementaria (por ejemplo, la dirección del cliente) que luego puede ser eliminada del archivo de datos. Observe que por lo general los archivos del catálogo son pequeños en comparación con el archivo de datos.

Esta organización es más eficiente en términos de espacio y E/S que el diseño original (el cual involucra un solo archivo de datos). Observe en particular, que la información sobre productos, periodos y clientes del archivo central de datos ahora se reduce a sólo *identificadores* de producto, periodo y cliente.

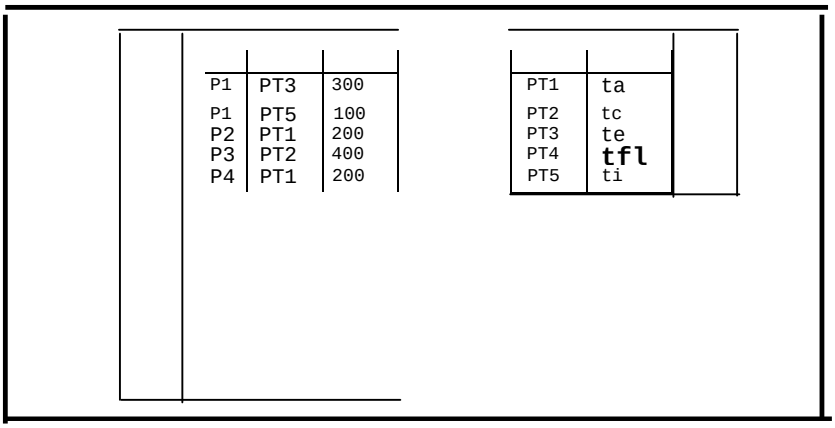
Cuando se emula este enfoque en una base de datos relacional, el archivo de datos y los archivos de catálogo se convierten en tablas (imágenes de los archivos correspondientes); los apuntadores que están en los archivos de catálogo se convierten en claves primarias en las tablas que son imagen de estos archivos, y los identificadores que están en el archivo de datos se convierten en claves externas en la tabla que es imagen del archivo de datos. Por lo general, estas claves primarias y externas están indexadas. Bajo tal arreglo, a la imagen del archivo de datos se le llama **tabla de hechos** y a las imágenes de los archivos de catálogo se les llama **tablas de dimensión.** Al diseño general se le menciona como **esquema dimensional,** o **de estrella,** debido a la forma en que aparece cuando es trazado como diagrama de entidad-vínculo (donde la tabla de hechos está rodeada por y conectada a las tablas de dimensión). *Nota:* En la sección 21.6 explicamos la razón de la terminología de "dimensión".

A manera de ejemplo, modifiquemos nuevamente la base de datos de proveedores y partes; esta vez para mostrar el periodo de tiempo particular en que sucedió cada envío. Identificamos a los periodos mediante un identificador de periodo de tiempo (PT#) e introducimos otra tabla PT para relacionar esos identificadores con los periodos de tiempo correspondientes. Entonces la tabla de envíos VP modificada y la nueva tabla de periodos PT pueden verse como muestra la figura 21.3.* En la terminología del esquema de estrella, la tabla VP es la tabla de hechos y la tabla PT es una tabla de dimensión (y también lo son la tabla de proveedores-V y la tabla de partes P; vea la figura 21.4). *Nota:* Le recordamos nuevamente que en el capítulo 22 trataremos en detalle la cuestión general del manejo de datos para periodos de tiempo.

Por lo general, consultar una base de datos con esquema de estrella involucra el uso de las tablas de dimensión para encontrar todas las combinaciones de clave externa que son de interés, y luego el uso de esas combinaciones para acceder a la tabla de hechos. Suponiendo que los accesos a las tablas de dimensión y el subsecuente acceso a la tabla de hechos están integrados en una sola consulta, la mejor forma para implementar esa consulta es, por lo general, por medio de lo que se llama una *junta de estrella.* La "junta de estrella" es una estrategia específica de implementación de junta; difiere con respecto a las estrategias usuales, en que comienza deliberadamente calculando un producto cartesiano (concretamente el producto cartesiano de las tablas de dimensión). Como vimos en el capítulo 17, optimizadores de consultas tratan generalmente de evitar el cálculo de productos cartesianos [17.54], [17.55]; sin embargo, en este caso la formación en primer lugar del producto de las tablas de dimensión, mucho más pequeñas, y luego la utilización del resultado para realizar búsquedas basadas en índice sobre la tabla de hechos es casi siempre más eficiente que cualquier otra estrategia. De esto deducimos que los optimizadores originales requieren algo de reingeniería para manejar eficientemente las consultas de esquema de estrella.

---

*Las columnas DESDE y HASTA de la tabla PT contienen datos del tipo marca de tiempo. Por razones de simplicidad, en la figura no mostramos los valores reales de las marcas de tiempo, sino que en su lugar, los representamos simbólicamente.



P1	PT3	300
P1	PT5	100
P2	PT1	200
P3	PT2	400
P4	PT1	200

PT1	ta
PT2	tc
PT3	te
PT4	**tfl**
PT5	ti

2. Los esquemas de estrella en realidad son físicos y no lógicos, aunque se habla de ellos como si fueran lógicos. El problema es que en realidad en el enfoque del esquema de estrella no hay concepto de diseño lógico para distinguirlo con respecto al diseño físico.
3. El enfoque del esquema de estrella no siempre da como resultado un diseño físico legítimo (es decir, uno que conserve toda la información en un diseño lógico relacionalmente correc to). Esta limitación se hace más aparente conforme el esquema es más complejo.
4. Debido a que hay muy poca disciplina, los diseñadores incluyen a menudo varios tipos de hechos diferentes en la tabla de hechos. Como consecuencia, las filas y columnas de la tabla de hechos no tienen (por lo general) una interpretación uniforme. Lo que es más, por lo ge neral ciertas columnas sólo se aplican a determinados tipos de hechos, lo que implica que las columnas en cuestión deben permitir nulos. Y conforme son incluidos más y más tipos de hechos, la tabla se hace cada vez más difícil de mantener y comprender, y el acceso se hace cada vez menos eficiente. Por ejemplo, podríamos decidir modificar la tabla de envíos para llevar la cuenta de las compras de partes y también de los envíos de partes. Entonces necesitaríamos algún tipo de columna "indicadora" para que nos mostrara cuáles filas co rresponden a las compras y cuáles a los envíos. Por el contrario, un diseño adecuado crearía una tabla de hechos distinta para cada tipo de hecho distinto.
5. De nuevo, debido a la falta de disciplina, las tablas de dimensión también pueden llegar a ser no uniformes. Por lo general este error sucede cuando la tabla de hechos es usada para man tener datos que se refieren a diferentes niveles de agregación. Por ejemplo, podríamos (erró neamente) añadir filas a la tabla de envíos para que mostraran las cantidades totales de partes para cada día, cada mes, cada trimestre, cada año e incluso, el gran total a la fecha. Observe primero que este cambio ocasiona que las columnas para el identificador de periodo (PT#) y la cantidad (CANT) de la tabla VP tengan significados no uniformes. Suponga ahora que las columnas DESDE y HASTA de la tabla de dimensión PT son reemplazadas por una com binación de columnas AÑO, MES, DÍA, etcétera. Entonces esas columnas AÑO, MES, DÍA, etcétera, deben ahora permitir nulos. También es probable que se necesite una columna indi cadora para que indique el tipo de periodo aplicado.
6. Con frecuencia, las tablas de dimensión no están normalizadas por completo.* El deseo de evitar juntas conduce frecuentemente a los diseñadores a agrupar en esas tablas informa ción distinta, que seria mejor mantener separada. En el caso extremo, las columnas a las que simplemente se tiene acceso en conjunto, son mantenidas juntas en la misma tabla de di mensión. Debe quedar claro que seguir tal "disciplina" extrema, y no relacional, conducirá con seguridad a una redundancia sin control y probablemente incontrolable.

Finalmente, comentamos que una variante del esquema de estrella es el esquema de **copo de nieve,** el cual normaliza las tablas de dimensión. De nuevo, el nombre está derivado de la

*Éste es un consejo de un libro sobre data warehouses: "[Resístase] a la normalización... Los esfuerzos para normalizar cualquiera de las tablas en una base de datos dimensional, solamente para ahorrar espacio de disco, son una pérdida de tiempo... Las tablas de dimensión no deben ser normalizadas... Las tablas de dimensión normalizadas destruyen la habilidad para navegar" [21.21],

forma en que luce el esquema cuando es trazado como un diagrama de entidad-vínculo. Los términos *esquema de constelación* y *esquema de ventisca* (o de *tormenta de nieve)* también se han usado recientemente con los significados obvios (¿?).

PROCESAMIENTO ANALÍTICO EN LINEA

El término **OLAP** ("procesamiento analítico en línea") fue acuñado en un artículo escrito por Arbor Software Corp. en 1993 [21.10], aunque (como sucede con el término "data warehouse") el concepto es mucho más antiguo. Puede ser definido como "el proceso interactivo de crear, mantener, analizar y elaborar informes sobre datos" y es usual añadir que los datos en cuestión son percibidos y manejados como si estuvieran almacenados en un "arreglo multidimensional". Sin embargo, decidimos explicar primero las ideas en términos de tablas convencionales estilo SQL, antes de entrar en el tema de la representación multidimensional como tal.

El primer punto es que el procesamiento analítico requiere invariablemente, algún tipo de *agregación de datos,* por lo general en muchas formas diferentes (es decir, de acuerdo con muchos agrupamientos diferentes). De hecho, uno de los problemas fundamentales del procesamiento analítico es que la cantidad de agrupamientos posibles llega rápidamente a ser muy grande y los usuarios deben considerarlos todos o casi todos. Ahora bien, por supuesto que los lenguajes relacionales soportan tal agregación, pero cada consulta individual en un lenguaje de éstos produce como resultado una sola tabla (y todas las filas de esa tabla tienen la misma forma y el mismo tipo de interpretación). Por lo tanto, para obtener *n* agrupamientos distintos se requieren *n* consultas distintas y *n* tablas de resultados distintas. Por ejemplo, considere las siguientes consultas sobre la base de datos usual de proveedores y partes:

1. Obtener la cantidad total de envíos.
2. Obtener las cantidades totales de envíos por proveedor.
3. Obtener las cantidades totales de envíos por partes.
4. Obtener las cantidades totales de envíos por proveedor y parte. *Nota:* Por supuesto, la cantidad "total" para un proveedor dado y una parte dada es simplemente la cantidad real para ese proveedor y parte. El ejemplo sería más realista si usáramos, en vez de ello, la base de datos de proveedores, partes y proyectos. Sin embargo, nos mantendremos con proveedores y partes por razones de simplicidad.

Supongamos que sólo hay dos partes, P l y P2, y que la tabla de envíos luce de esta forma:

VP

V#	P#	CANT
V1	P1	300
V1	P2	200
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200

Entonces éstas son las formulaciones* SQL de las cuatro consultas y sus resultados correspondientes:

1.
```
SELECT SUM (CANT) AS CANTOT
FROM   VP
GROUP BY () ;
```

CANTOT
1600

2.
```
SELECT V#,
       SUM (CANT) AS CANTOT
FROM   VP GROUP BY (V#) ;
```

V#	CANTOT
V1	500
V2	700
V3	200
V4	200

3.
```
SELECT P#,
       SUM(CANT) AS CANTOT
FROM   VP
GROUP BY (P#) ;
```

P#	CANTOT
P1	600
P2	1000

4.
```
SELECT V#, P#,
       SUM(CANT) AS CANTOT
FROM   VP GROUP BY (V#,
P#) ;
```

V#	P#	CANTOT
V1	P1	300
V1	P2	200
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200

Las desventajas de este enfoque son obvias: la formulación de tantas consultas similares, pero distintas, es tediosa para el usuario, y la ejecución de todas esas consultas —pasando una y otra vez por los mismos datos— es probablemente bastante costosa en tiempo de ejecución. Por lo tanto, parece que vale la pena tratar de encontrar una forma (a) de solicitar varios niveles de agregación en una sola consulta y por lo tanto, (b) ofrecer a la implementación la oportunidad de calcular todas esas agregaciones de manera más eficiente (es decir, en un solo paso). Dichas consideraciones son la motivación que hay tras las opciones *GROUPING SETS, ROLLUP y CUBE* de la cláusula GROUP BY. *Nota:* Dichas opciones ya son soportadas en varios productos comerciales. También están incluidas en SQL3 (vea el apéndice B).

La opción **GROUPING SETS** permite al usuario especificar con exactitud qué agrupamientos específicos van a ser realizados. Por ejemplo, la siguiente instrucción SQL representa una combinación de las consultas 2 y 3:

```
SELECT V#, P#, SUM (CANT) AS CANTOT
FROM   VP
GROUP BY GROUPING SETS ( (V#), (P#) ) ;
```

Aquí la cláusula GROUP BY está pidiendo efectivamente al sistema que ejecute dos consultas, una donde el agrupamiento sea por V# y otra en la que sea por P#. *Nota:* Los paréntesis **inte-**

*Tal vez debiéramos decir "formulaciones *seudoSQL*", ya que el SQL/92 no permite que los operandos de GROUP BY estén encerrados entre paréntesis, ni permite que GROUP BY no tenga operandos (aunque la omisión completa de la cláusula GROUP BY es lógicamente equivalente a la especificación de una sin operandos).

riores en realidad no son necesarios en este ejemplo, ya que cada uno de los "conjuntos de agrupamiento" involucra sólo una columna, pero los mostramos por razones de claridad.

Ahora bien, la idea de agrupar de esta manera varias consultas diferentes en una sola instrucción puede ser inobjetable por sí misma (aunque tenemos que decir que preferiríamos atacar este tema muy general en una forma aún más general, sistemática y ortogonal). Sin embargo, por desgracia ¡el SQL continúa agrupando los *resultados* de estas consultas lógicamente distintas en una sola tabla de resultados!* En el ejemplo, esa tabla de resultados luce así:

V#	P#	CANTOT
V1	*nulo*	500
V2	*nulo*	700
V3	*nulo*	200
V4	*nulo*	200
nulo	P1	600
nulo	P2	1000

Ahora bien, este resultado puede ser visto como una *tabla* (una tabla SQL, de alguna forma), pero difícilmente es una *relación.* Observe que las filas de proveedor (las que tienen nulos en la posición P#) y las filas de partes (las que tienen nulos en la posición V#) tienen interpretaciones muy diferentes y el significado de los valores de CANTOT varía según aparezca en una fila de proveedor o en una fila de parte. Entonces, ¿cuál es el predicado para esta "relación"?

Observamos también que los nulos en este resultado constituyen otro tipo de "información faltante". En realidad no significan "valor desconocido" ni "valor no aplicable", pero lo que significan exactamente es muy confuso. *Nota:* SQL proporciona al menos una forma para distinguir esos nuevos nulos con respecto a otros tipos, pero los detalles son tediosos y fuerzan al usuario hacia un tipo de pensamiento de fila por fila. Aquí omitimos esos detalles, pero puede obtener alguna idea de lo que está implícito a partir del siguiente ejemplo (que indica cómo podría aparecer realmente en la práctica el ejemplo de GROUPING SETS que mostramos anteriormente):

```
SELECT CASE GROUPING (V#)                    -- vea la referencia a CASE en el apéndice A
       WHEN 1 THEN '?????'
       ELSE V# AS V#, CASE
       GROUPING (P#)
       WHEN 1 THEN '!!!!!!' ELSE             -- vea la referenda a CASE en el apéndice A
       P# AS P#,
       SUM (CANT) AS CANTOT FROM  VP
GROUP BY GROUPING SETS ( (V#) , (P#)
```

Regresemos específicamente a GROUP BY. Las otras dos opciones de GROUP BY (ROLLUP y CUBE) son abreviaturas para ciertas combinaciones de GROUPING SETS. Primero, veamos **ROLLUP.** Considere la siguiente consulta:

---

*Esta única tabla puede ser considerada como una "unión externa" —también, una forma extremadamente rara de unión externa— de esos resultados. Debe quedar claro, por lo que dijimos en el capítulo 18, que aun en su forma menos rara, la "unión externa" *no* es una operación relacional respetable.

```
SELECT V#, P#, SUM(CANT) AS CANTOT
FROM   VP
GROUP BY ROLLUP ( V#, P# ) ;
```

Aquí la cláusula GROUP BY es lógicamente equivalente a la siguiente:

```
GROUP BY GROUPING SETS ( (V#,P#), (V#), () )
```

En otras palabras, la consulta es una formulación SQL combinada de las consultas 4, 2 y 1. El resultado se ve de esta forma:

V#	P#	CANTOT
V1	P1	300
V1	P2	200
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200
V1	*nulo*	500
V2	*nulo*	700
V3	*nulo*	200
V4	*nulo*	200
nulo	*nulo*	1600

El termino ROLLUP se deriva del hecho que (en el ejemplo) las cantidades han sido "enrolladas" para cada proveedor (es decir, enrolladas "en toda la dimensión de proveedor"; vea la subsección "Bases de datos multidimensionales" que sigue a continuación). En general, GROUP BY ROLLUP (A, *B,* ..., Z) —aproximadamente, "enrollar en toda la dimensión A"— significa "agrupar por todas las combinaciones siguientes":

```
( A, B, .     Z )
( A, B, .
```

```
( A,  B )
( A )
```

Observe que hay muchos "enrollar en toda la dimensión A" distintos, en general (depende de qué otras columnas son mencionadas en la lista de ROLLUPs separados con comas). Observe también que GROUP BY ROLLUP (A, *B)* y GROUP BY ROLLUP (ñ, A) tienen significados diferentes; es decir, GROUP BY ROLLUP (A, *B) no* es simétrico en A y *B.* Pasemos ahora a **CUBE.** Considere la siguiente consulta:

```
SELECT V#, P#, SUM(CANT) AS CANTOT
FROM   VP
GROUP  BY CUBE  ( V#,  P# ) ;
```

Aquí la cláusula GROUP BY es lógicamente equivalente a la siguiente:

```
GROUP BY GROUPING SETS ( (V#,P#), (V#), (P#), () )
```

En otras palabras, la consulta es una formulación SQL combinada de las cuatro consultas originales 4, 3, 2 y 1. El resultado se ve de esta forma:

V#	P#	CANTOT
V1	P1	300
V1	P2	200
V2	P1	300
V2	P2	400
V3	P2	200
V4	P2	200
V1	*nulo*	500
V2	*nulo*	700
V3	*nulo*	200

El término CUBE poco útil, se deriva del hecho de que en la terminología OLAP (o al menos multidimensional), los valores de datos pueden ser percibidos como si estuvieran almacenados en las celdas de un arreglo multidimensional o *hipercubo*. En el caso que estamos viendo (a) los valores de datos son cantidades, (b) el "cubo" es de dos dimensiones: una dimensión de proveedores y una dimensión de partes (¡y el "cubo" es bastante plano!) y por supuesto, (c) esas dos dimensiones son de tamaños desiguales (por lo que el "cubo" ni siquiera es un cuadrado, sino un rectángulo general). De cualquier forma, GROUP BY CUBE (A, *B*,..., Z) significa "agrupar por todos los subconjuntos posibles del conjunto *{A, B*,..., Z}".

Una cláusula GROUP BY dada puede incluir cualquier mezcla de especificaciones GROUPING SETS, ROLLUP y CUBE.

Tabulaciones cruzadas

Con frecuencia, los productos OLAP despliegan los resultados de las consultas no como tablas estilo SQL sino como **tabulaciones cruzadas** (abreviado "tabcruz"). Considere nuevamente la consulta 4 ("obtener el total de envíos por proveedor y parte"). Ésta es una representación tabcruz del resultado de esa consulta. Dicho sea de paso, observe que mostramos las cantidades de la parte Pl para los proveedores V3 y V4 (correctamente) como cero; por el contrario, SQL diría que estas cantidades deben ser *nulos* (vea el capítulo 18). De hecho, la *tabla* que produce SQL en respuesta a la consulta 4 ¡no contiene filas para (V3, Pl) ni para (V4, Pl)! Como consecuencia de esto, la producción de la tabcruz a partir de la tabla no es completamente trivial.

V4	*nulo*	200
nulo	P1	600
nulo	P2	1000
nulo	*nulo*	1600

Podemos decir que esta tabcruz es una forma más compacta y legible de representar el resultado de la consulta 4. Además, se parece más a una tabla relacional. Sin embargo, observe que *la cantidad de columnas en esa "tabla" depende de los datos reales;* para ser más específicos,

	P1	P2
V1	300	200
V2	300	400
V3	0	200
V4	0	200

hay una columna para cada tipo de parte (y por lo tanto, la estructura de la tabcruz y el significado de las filas dependen de los datos reales). Por lo tanto, una tabcruz no es una relación sino un *informe;* para ser más específicos, un informe que tiene el formato de un arreglo simple. (Las relaciones tienen un predicado que puede deducirse a partir de los predicados de las relaciones de las cuales están derivadas; por el contrario, el "predicado" de una tabcruz —si es que en general podemos decir que existe— no puede deducirse de los predicados de las relaciones de las cuales está derivada, sino que, como ha visto, depende de los *valores* de los datos.)

Con frecuencia decimos que las tabcruz como la que acabamos de mostrar, tienen dos *dimensiones:* en este caso proveedores y partes. Las dimensiones son tratadas como si fueran *variables independientes;* entonces, las "celdas" de intersección contienen los valores de las variables *dependientes* correspondientes. Para una explicación más a fondo, vea la subsección "Bases de datos multidimensionales" que sigue.

Éste es otro ejemplo de tabcruz que representa el resultado del ejemplo anterior de CUBE:

	P1	P2	*total*
V1	300	200	500
V2	300	400	700
V3	0	200	200
V4	0	200	200
total	600	1000	1600

La columna del extremo derecho contiene totales de filas (es decir, los totales para el proveedor indicado en todas las partes) y la fila inferior contiene totales de columnas (es decir, los totales para la parte indicada en todos los proveedores). La celda de la parte inferior derecha contiene el gran total, que es el total de fila de todos los totales de columna y el total de columna de todos los totales de fila.

Bases de datos multidimensionales

Hasta ahora hemos estado dando por hecho que nuestros datos OLAP están almacenados en una base de datos SQL convencional (aunque hemos mencionado la terminología y los conceptos de las bases de datos "multidimensionales" unas cuantas veces). De hecho, hemos estado describiendo tácitamente lo que en ocasiones se llama *ROLAP* ("OLAP relational"). Sin embargo, mucha gente cree que el *MOLAP* ("OLAP multidimensional") es una mejor forma. En esta sección daremos un vistazo más cercano al MOLAP.

El MOLAP involucra una **base de datos multidimensional,** que es una base de datos en la cual los datos están almacenados conceptualmente en las celdas de un arreglo multidimensional. *(Nota:* Decimos almacenados "conceptualmente", pero de hecho, la organización física de MOLAP tiende a ser muy similar a la organización lógica.) El DBMS que lo soporta se llama *DBMS multidimensional.* Como un ejemplo simple, los datos podrían estar representados como un arreglo de tres dimensiones que corresponden a productos, clientes y periodos, respectivamente; cada valor individual de celda podría representar la cantidad total del producto indicado, vendido al cliente indicado en el periodo indicado. Como ya dijimos, las tabcruz de la subsección anterior también pueden ser consideradas como dichos arreglos.

Ahora bien, en un cuerpo de datos bien comprendido, todos los vínculos serían conocidos y las "variables" involucradas (no las variables en el sentido usual de los lenguajes de programación)

podrían ser clasificadas en términos generales como **dependientes** o **independientes.** En términos del ejemplo anterior, *producto, cliente* y *periodo* serían las variables independientes, mientras que *cantidad* sena la única variable dependiente. De forma más general, las variables independientes son aquellas cuyos valores determinan en conjunto los valores de las variables dependientes (de forma similar a como en términos relaciónales, una clave candidata es un conjunto de columnas cuyos valores determinan los valores de otras columnas). Por lo tanto, las variables independientes forman las dimensiones del arreglo con el que están organizados los datos y forman un *esquema de direccionamiento* para ese arreglo;* y los valores de la variable dependiente —que constituyen los datos reales— pueden entonces ser almacenados en las celdas de ese arreglo. *Nota:* La diferencia entre los valores de las variables independientes (o "dimensionales") y los valores de las variables dependientes (o "no dimensionales") se caracterizan en ocasiones como *ubicación* contra *contenido.*

Por desgracia, la caracterización anterior de las bases de datos multidimensionales es en cierta forma demasiado simplista, ya que la mayoría de los cuerpos de los datos *no* están bien entendidos. Además, ésta es la razón principal por la que queremos analizar en primer lugar los datos: para lograr una mejor comprensión. Con frecuencia la falta de comprensión es tan grande que ni siquiera sabemos de antemano cuáles variables son independientes y cuáles son dependientes; a menudo las variables independientes son seleccionadas con base en las creencias actuales (es decir, *hipótesis)* y entonces se prueba el arreglo resultante para ver qué tan bien funciona (vea la sección 21.7). Dicho enfoque involucrará claramente muchas iteraciones y tanteos. Por tales razones, el sistema generalmente permitirá intercambiar las variables dimensionales y no dimensionales, una operación conocida como *pivoteo.* Otras operaciones soportadas incluirán la *transposición de arreglos* y el *reordenamiento dimensional.* También habrá formas de añadir dimensiones.

A propósito, a partir de la descripción anterior debe quedar claro que las celdas del arreglo estarán frecuentemente vacías y por lo tanto, los arreglos serán *poco densos.* Por ejemplo, suponga que el producto *p* no fue vendido al cliente *c* durante el periodo *t.* Entonces la celda *(c,p,t)* estará vacía (o en el mejor caso, contendrá cero). Los DBMSs multidimensionales soportan diversas técnicas para el almacenamiento de arreglos poco densos en alguna forma más eficiente (comprimida), t Además, esas celdas vacías corresponden a "información faltante" y por lo tanto, los sistemas deben proporcionar algún soporte computacional para ellas; lo cual (desafortunadamente) lo hacen generalmente en forma similar a SQL. Observe que el hecho de que una celda esté vacía puede significar que la información es desconocida, que no ha sido capturada, que no es aplicable o muchas cosas más (vea nuevamente el capítulo 18).

Con frecuencia, las variables independientes están relacionadas en *jerarquías,* las cuales determinan las formas en que es posible agregar los datos dependientes. Por ejemplo, hay una jerarquía temporal que relaciona segundos con minutos, con horas, con días, con semanas, con meses, con años. Como otro ejemplo, puede haber una jerarquía que relacione partes con conjuntos ensamblados con componentes, con tableros, con productos. A menudo los propios datos pueden

---

*Por lo tanto, las celdas del arreglo son referidas simbólicamente, en lugar de hacerlo por medio de los subíndices numéricos que están asociados más convencionalmente con los arreglos. [1]Observe el contraste con los sistemas relaciónales. En una analogía relacional adecuada para el ejemplo, no tendríamos una fila *(c,p,t)* con una "celda" vacía, sino que simplemente no tendríamos una fila *(c,p,i).* Por lo tanto, no se presenta el concepto de "arreglos poco densos" o "tablas poco densas", por lo que no hay necesidad de manejar técnicas de compresión inteligentes.

ser agregados en muchas formas diferentes (es decir, las mismas variables independientes pueden pertenecer a varias jerarquías diferentes). El sistema proporcionará operadores para "subir" y "bajar" por dichas jerarquías; donde *subir* significa ir de un nivel menor de agregación hacia uno más alto y *bajar* significa lo contrario. También existen muchas otras operaciones para manejar dichas jerarquías (por ejemplo, una operación para reacomodar los niveles de jerarquía).

Nota: Existe una diferencia sutil entre "subir" y "enrollar", que es la siguiente: "enrollar" es la operación para *crear* los agolpamientos y agregaciones deseadas; "subir" es la operación para *acceder* a esas agregaciones. Un ejemplo para "bajar" podría ser: *dadas las cantidades totales de envíos, obtener las cantidades totales para cada proveedor individual.* Por supuesto, es necesario tener (o poder calcular) los datos más detallados para poder responder a una solicitud de éstas.

Por lo general, los productos multidimensionales también proporcionan una variedad de funciones estadísticas y matemáticas para ayudar en la formulación y prueba de las hipótesis (por ejemplo, los vínculos hipotéticos). También se proporcionan herramientas de visualización y de informes para que ayuden en estas tareas. Sin embargo, por desgracia todavía no existe un lenguaje de consulta multidimensional estándar, aunque se están realizando investigaciones para desarrollar un cálculo sobre el cual pueda estar basado un estándar [21.27]. Asimismo, no existe algo similar a la teoría de la normalización que pudiera servir como base científica para el diseño de bases de datos multidimensionales.

Cerramos esta sección comentando que algunos productos combinan los enfoques ROLAP y MOLAP en el *HOLAP* ("OLAP híbrido"). Hay muchas controversias sobre cuál de estos tres enfoques es el "mejor" y aquí poco podemos decir para ayudar a resolver esa controversia.* Sin embargo, en términos generales los productos MOLAP proporcionan cálculos más rápidos pero soportan cantidades de datos más pequeñas que los productos ROLAP (con lo que llegan a ser menos eficientes conforme se incrementa la cantidad de datos) y en cambio, los productos ROLAP proporcionan características de escalabilidad, concurrencia y administración que son más maduras que las de los productos MOLAP.

21.7 MINERÍA DE DATOS

La **minería de datos** puede describirse como "análisis de datos exploratorio". El propósito es buscar patrones interesantes en los datos, patrones que pueden usarse para especificar la estrategia del negocio o para identificar comportamientos fuera de lo común (por ejemplo, un incremento súbito en la actividad de una tarjeta de crédito puede indicar que la tarjeta ha sido robada). Las herramientas de minería de datos aplican técnicas estadísticas a una gran cantidad de datos almacenados para buscar tales patrones. *Nota:* Aquí es necesario enfatizar la palabra **gran.** Las bases de datos para minería de datos frecuentemente son *extremadamente* grandes, y es importante que los algoritmos sean escalables.

---

*Sin embargo, hay algo que debemos decir. A menudo se afirma que "las tablas son planas" (es decir, de dos dimensiones); en cambio, "los datos reales son multidimensionales" y por lo tanto, las relaciones son inadecuadas como base para OLAP. Pero dar estos argumentos ;es confundir las tablas y las relaciones! Como vimos en el capítulo 5, las tablas son sólo *imágenes* de relaciones y no relaciones como tales. Y aunque es cierto que esas imágenes son de dos dimensiones, las relaciones no lo son; en su lugar son *n* dimensionales, donde *n* es el grado. Para ser más precisos, cada tupia en una relación con *n* atributos representa un punto en un espacio *n* dimensional y la relación como un todo, representa un conjunto de dichos puntos.

VENTAS

TX#	CLIENTE	MARCA DE TIEMPO	PRODUCTO
TX1	C1	tft	Zapatos
TX1	C1	*ai*	Calcetines
TX1	C1	*di*	Corbata
TX2	C2	*d2*	Zapatos
TX2	C2	*62*	Calcetines
TX2	C2	*02*	Corbata
TX2	C2	*02*	Cinturón
TX2	C2	*d2*	Camisa
TX3	C3	*d2*	Zapatos
TX3	C3	*d2*	Corbata
TX4	C2	*03*	Zapatos
TX4	C2	*d3*	Calcetines
TX4	C2	*d3*	Cinturón

Figura 21.5 La tabla VENTAS.

Considere la tabla VENTAS (¡que *no* es muy grande!) de la figura 21.5, la cual muestra información con respecto a ciertas transacciones de ventas de un negocio al menudeo.* Al negocio le gustaría realizar un *análisis de canasta de mercado* sobre estos datos (donde el término "canasta de mercado" se refiere al conjunto de productos comprados en una sola transacción) para descubrir de ese modo que, por ejemplo, es probable que un cliente que compra zapatos también compre calcetines como parte de la misma transacción. Esta correlación entre zapatos y calcetines es un ejemplo de una **regla de asociación;** puede ser expresada (en términos generales) de la siguiente forma:

```
FORALL íx ( Zapatos e íx ⇒■ Calcetines e tx )
```

(Donde "Zapatos e *tx"* es la *regla antecedente,* "Calcetines e *tx"* es la *regla consecuente* y *tx* abarca todas las transacciones de ventas.)

Presentamos algo de terminología. Al conjunto de todas las transacciones de ventas en el ejemplo se le llama la **población.** Cualquier regla de asociación tiene un nivel de *soporte* y un nivel de *confianza.* El **soporte** es el fragmento de la población que satisface la regla. La confianza es la fracción de esa parte de la población en la cual, si se satisface el antecedente, también se satisface el consecuente. (Observe que el antecedente y el consecuente pueden involucrar cada uno cualquier cantidad de productos diferentes y no necesariamente uno solo.) A manera de ejemplo considere esta regla:

```
FORALL tx ( Calcetines e íx ⇒• Corbatas c tx )
```

Dados los datos de ejemplo de la figura 21.5, la población es 4, el soporte es 50 por ciento y la confianza es del 66.67 por ciento.

Observe que (a) la clave es {TX#, PRODUCTO}; (b) la tabla satisface las DFs TX# -» CLIENTE y TX# —>MARCA DE TIEMPO, y por lo tanto, no está en FNBC; (c) una versión de la tabla en la cual la columna PRODUCTO tuviera valores de relación (y TX# fuera la clave) *estaría* en FNBC y podría ser más adecuada para el tipo de exploración involucrada en este caso (pero tal vez no sería adecuada para otros tipos).

Podemos descubrir reglas de asociación más generales a partir de agregaciones adecuadas de los datos dados. Por ejemplo, el agrupamiento por cliente nos permitiría probar la validez de reglas tales como "si un cliente compra zapatos es probable que también compre calcetines, aunque no necesariamente en la misma transacción".

También podemos definir otros tipos de reglas. Por ejemplo, una regla de **correlación de secuencia** podría ser usada para identificar patrones de compra a lo largo del tiempo ("si un cliente compra zapatos hoy, es probable que compre calcetines dentro de los cinco días siguientes"). Una regla de clasificación podría ser usada para ayudar a decidir si se otorga un crédito ("si un cliente tiene ingresos superiores a $75,000 anuales, es probablemente un buen sujeto de crédito") y así sucesivamente. Al igual que las reglas de asociación, las reglas de correlación de secuencia y de clasificación también tienen un nivel de soporte y un nivel de confianza.

La minería de datos es un tema inmenso por derecho propio [21.1] y es claro que aquí no es posible entrar en mucho detalle. Por lo tanto, nos contentamos con una breve descripción final sobre la manera en que es posible aplicar las técnicas de minería de datos en una versión extendida de los proveedores y partes. Primero (en ausencia de otras fuentes de información) podríamos usar la *inducción neural* para clasificar a los proveedores de acuerdo con su especialidad (por ejemplo, sujetadores contra partes de motor) y la *predicción de valor* para predecir cuáles son los proveedores que tienen mayor probabilidad de proporcionar determinadas partes. Luego podríamos usar el *agrupamiento demográfico* para asociar los cargos de envío con la ubicación geográfica para asignar, por lo tanto, los proveedores a las regiones de envíos. El *descubrimiento de asociación* podría ser usado entonces para descubrir que determinadas partes son adquiridas por lo general en un solo envío; el *descubrimiento del patrón secuencial* para descubrir que los envíos de sujetadores están seguidos generalmente de envíos de partes de motor; y el *descubrimiento de la secuencia de tiempo similar* para descubrir que hay cambios cuantitativos estacionales en los envíos de determinadas partes (algunos de esos cambios suceden en otoño y otros en primavera).

21.8 RESUMEN

Hemos examinado el uso de la tecnología de base de datos para efectos del **apoyo a la toma de decisiones.** La idea básica es recolectar datos operacionales y reducirlos a una forma que pueda ser utilizada para que la administración entienda y modifique el comportamiento de la empresa.

Primero identificamos ciertos aspectos de los sistemas de apoyo para la toma de decisiones que los apartan de los sistemas operacionales. El punto principal es que la base de datos es en su mayoría (aunque no totalmente) de **sólo lectura.** Las bases de datos de apoyo para la toma de decisiones tienden a ser muy **grandes** y **altamente indexadas** —e involucran una gran cantidad de **redundancia controlada** (en especial en la forma de *replicación* y *agregaciones* precalculadas)—, las claves tienden a involucrar un componente **temporal** y las consultas tienden a ser **complejas.** Como consecuencia de tales consideraciones, existe un énfasis en el **diseño para el rendimiento;** estamos de acuerdo con el énfasis, pero creemos que no debemos permitir que interfiera con las buenas prácticas de diseño. El problema es que en la práctica, los sistemas de apoyo para la toma de decisiones no distinguen adecuadamente (por lo general) entre las consideraciones **lógicas** y las **físicas.**

Después explicamos lo que está involucrado en la preparación de los datos operacionales para el apoyo a la toma de decisiones. Vimos las tareas de **extracción, limpieza, transformación y consolidación, carga** y **actualización.** También mencionamos brevemente a los **almacenes de datos operacionales,** que pueden ser usados (entre otras cosas) como área transitoria durante

el proceso de preparación de datos. También pueden ser usados para proporcionar servicios de apoyo para la toma de decisiones sobre datos actuales.

Luego consideramos los **data warehouses** y los **data marts;** un data mart puede ser considerado como un data warehouse especializado. Explicamos la idea básica de los **esquemas de estrella,** en los cuales los datos están organizados como una gran **tabla de hechos** central y varias **tablas de dimensión** mucho más pequeñas. En situaciones simples, un esquema de estrella no es distinguible con respecto a un esquema normalizado clásico; sin embargo, en la práctica los esquemas de estrella se apartan de los principios de diseño clásicos en varias formas, siempre por razones de rendimiento. (De nuevo, el problema es que por naturaleza, los esquemas de estrella en realidad son más físicos que lógicos.) También mencionamos la estrategia de implementation de junta conocida como *junta de estrella* y una variante del esquema de estrella llamada esquema de **copo de nieve.**

Luego enfocamos nuestra atención en el **OLAP.** Tratamos las características **GROUPING SETS, ROLLUP** y **CUBE** de SQL (todas son opciones de la cláusula GROUP BY y proporcionan formas para solicitar varias agregaciones distintas dentro de una sola consulta SQL). Observamos que SQL (desafortunadamente, en nuestra opinión) agrupa los resultados de esas agregaciones distintas en una sola "tabla" que contiene gran cantidad de nulos. También sugerimos que en la práctica, los productos OLAP podrían convertir dichas "tablas" en **tablas cruzadas** (arreglos simples) para efectos de presentación. Luego dimos un vistazo a las **bases de datos multídimensionales** en las cuales los datos están almacenados, conceptualmente, no en tablas sino en un arreglo multidimensional o *hipercubo.* Las dimensiones de dicho arreglo representan **variables independientes** y las celdas contienen valores de las **variables dependientes** correspondientes. Por lo general, las variables independientes están relacionadas en diversas **jerarquías,** las cuales determinan la forma en que los datos dependientes pueden ser agrupados y agregados.

Por último consideramos la **minería de datos.** Aquí la idea básica es que debido a que los datos de apoyo para la toma de decisiones con frecuencia no son muy bien entendidos, podemos usar el poder de la computadora para que nos ayude a descubrir patrones en los datos. Consideramos brevemente varios tipos de *reglas* —de **asociación,** clasificación y **correlación de secuencia**— y tratamos las nociones asociadas de niveles de **soporte** y de **confianza.**

[ERCICIOS

21.1 ¿Cuáles son algunos de los puntos principales de diferencia entre las bases de datos de apoyo para la toma de decisiones y las operacionales? ¿Por qué las aplicaciones de apoyo para la toma de decisiones y las aplicaciones operacionales utilizan generalmente almacenes de datos diferentes?

21.2 Resuma los pasos involucrados en la preparación de datos operacionales para el apoyo a la toma de decisiones.

21.3 Distinga entre la redundancia *controlada* y la *no controlada.* Dé algunos ejemplos. ¿Por qué es importante la redundancia controlada en el contexto del apoyo para la toma de decisiones? ¿Qué sucede si, en vez de ello, la redundancia no es controlada?

21.4 Distinga entre *data warehouses y data marts.*

21.5 ¿Qué entiende usted por el término *esquema de estrella!*

21.6 Por lo general, los esquemas de estrella no están completamente normalizados. ¿Cuál es la justi ficación para esta situación? Explique la metodología mediante la cual son diseñados dichos esquemas.

21.7 Explique la diferencia entre ROLAP y MOLAP.

21.8 ¿En cuántas formas es posible resumir los datos si están caracterizados por cuatro dimensiones, donde cada una de ellas pertenece a una jerarquía de agregación de tres niveles (por ejemplo, ciudad, región, estado)?

21.9 Utilice la base de datos de proveedores, partes y proyectos (vea el ejercicio 4.1 del capítulo 4) para expresar lo siguiente como consultas SQL:

a. Obtener la cantidad de envíos y las cantidades promedio de envíos para proveedores, partes y pro yectos considerados en pares (es decir, para cada par V#-P#, cada par P#-Y# y cada par Y#-V#).
b. Obtener las cantidades máxima y mínima de envíos para cada proyecto, para cada combinación de proyecto y parte, y en general.
c. Obtener las cantidades de envíos totales enrolladas "en toda la dimensión de proveedor" y "en toda la dimensión de parte". *Precaución:* Aquí hay una trampa.
d. Obtener las cantidades de envío promedio por proveedor, por parte, por combinaciones de pro veedor/parte y en general.

En cada caso, muestre el resultado SQL que se produciría tomando los datos de ejemplo de la figura 4.5 (o algunos datos de ejemplo que usted proporcione). También muestre esos resultados como tablas cruzadas.

21.10 Casi al principio de la sección 21.6, mostramos una versión simple de la tabla VP que contiene solamente seis filas. Suponga que la tabla incluye además la siguiente fila (lo que significa que ¡tal vez! existe el proveedor V5 pero actualmente no proporciona partes):

V5	*nulo*	*nulo*

Explique las implicaciones para todas las diversas consultas SQL que mostramos en la sección **21.6.**

21.11 ¿Significa lo mismo el término *dimensional* en las frases "esquema dimensional" y "base de datos multidimensional"? Explique su respuesta.

21.12 Considere el problema del análisis de canasta de mercado. Indique un algoritmo mediante el cual sea posible descubrir las reglas de asociación que tienen niveles de soporte y de confianza ma yores que los límites especificados. *Sugerencia:* Si alguna combinación de productos "no es intere sante" debido a que sucede en muy pocas transacciones de ventas, entonces lo mismo es cierto para todos los superconjuntos de esa combinación de productos.

REFERENCIAS Y BIBLIOGRAFÍA

21.1 Pieter Adriaans y Dolf Zantinge: *Data Mining*. Reading, Mass.: Addison-Wesley (1996).

Aunque es descrito como una panorámica en el nivel ejecutivo, este libro es en realidad una in troducción bastante detallada (y buena) al tema.

21.2 S. Alter: *Decision Support Systems: Current Practice and Continuing Challenges.* Reading, Mass.: Addison-Wesley (1980).

21.3 J. L. Bennett (ed.): *Building Decision Support Systems.* Reading, Mass.: Addison-Wesley (1981).

21.4 M. J. A. Berry y G. Linoff: *Data Mining Techniques for Marketing, QTY, and Customer Support.* Nueva York, N.Y.: McGraw-Hill (1997).

Es una buena explicación sobre los métodos de la minería de datos y su valor para aspectos se- leccionados de los negocios.

21.5 J. B. Boulden: *Computer-Assisted Planning Systems.* Nueva York, N.Y.: McGraw-Hill (1975)

Este primer texto trata muchas de las preocupaciones que posteriormente fueron reunidas bajo el encabezado de apoyo para la toma de decisiones. Como lo dice el título, el énfasis está en la planeación de la administración en el sentido clásico.

21.6 R. H. Bonczek, C. W. Holsapple y A. Whinston: *Foundations of Decision Support Systems.* Orlando, Fla.: Academic Press (1981).

Es uno de los primeros textos en promover un enfoque disciplinado para los sistemas de apoyo para la toma de decisiones. Enfatiza los papeles del modelado (en el sentido general de modelado empírico y matemático) y la ciencia de la administración.

21.7 Charles J. Bontempo y Cynthia Maro Saracco: *Database Management: Principles and Products.* Upper Saddle River, N.J.: Prentice-Hall (1996).

21.8 P. Cabena, P. Hadjinian, R. Stadler, J. Verhees y A. Zanasi: *Discovering Data Mining: From Concept to Implementation.* Upper Saddle River, N.J.: Prentice-Hall (1998).

21.9 C. L. Chang: "DEDUCE—A Deductive Query Language for Relational Data Bases", en C. H. Chen (ed.), *Pattern Recognition and Artificial Intelligence.* Nueva York, N.Y.: Academic Press (1976).

21.10 E. F. Codd, S. B. Codd y C. T. Salley: "Providing OLAP (Online Analytical Processing) to User-Analysts: An IT Mandate", disponible con Arbor Software Corp. (1993).

Como mencionamos en el cuerpo del capítulo, este artículo es el origen del término "OLAP" (aunque no del concepto). Es interesante observar que casi al comienzo, el artículo establece categóricamente que "la necesidad existente NO es de otra tecnología de base de datos sino, en su lugar, de robustas... herramientas de análisis". ¡Después continúa describiendo y argumentando sobre la necesidad de otra tecnología de base de datos!, con una nueva representación conceptual de datos, nuevos operadores (para actualización y también para recuperación), soporte multiusuarios (incluyendo características de seguridad y concurrencia), nuevas estructuras de almacenamiento y nuevas características de optimización; en otras palabras, un nuevo modelo de datos y un nuevo DBMS.

21.11 C. **J.** Date: "We Don't Need Composite Columns", en C. J. Date, Hugh Darwen y David McGoveran, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).

El título de este artículo se refiere al hecho de que en el pasado se ha intentado (sin éxito) introducir soporte para columnas compuestas, sin basarlo en el soporte para los tipos definidos por el usuario. Si se proporciona soporte adecuado para los tipos definidos por el usuario, las columnas compuestas "saldrán bien".

21.12 Barry Devlin: *Data Warehouse from Architecture to Implementation.* Reading, Mass.: Addison-Wesley (1997).

21.13 B. A. Devlin y P. T. Murphy: "An Architecture for a Business and Information System", *IBM Sys. J. 27,* No. 1 (1988).

Es el primer artículo publicado que define y usa el término "almacén de información".

21.14 Herb Edelstein: *Data Mining: Products and Markets.* Potomac, Md.: Two Crows Corp. (1997).

21.15 T. P. Gerrity, Jr.: "The Design of Man-Machine Decision Systems: An Application to Portfo lio Management", *Sloan Management Review 12,* No. 2 (invierno, 1971).

Es uno de los primeros artículos sobre los sistemas de apoyo para la toma de decisiones. Describe un sistema para dar apoyo a los administradores de inversiones en la administración de portafolios de acciones.

21.16 Jim Gray, Adam Bosworth, Andrew Layman y Hamid Pirahesh: "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", Proc. 12th IEEE Int. Conf. on Data Engineering, Nueva Orleans, La. (febrero, 1996).

Es el artículo que sugiere por primera vez la adición de opciones tales como CUBE a la cláusula GROUP BY de SQL.

21.17 W. H. Inmon: *Data Architecture: The Information Paradigm.* Wellesley, Mass.: QED Infor mation Sciences (1988).

Trata el origen del concepto data warehouse y cómo luciría un data warehouse en la práctica. El término "data warehouse" apareció por primera vez en este libro.

21.18 W. H. Inmon: *Building the Data Warehouse.* Nueva York, N.Y.: Wiley (1992).

Es el primer libro dedicado a los data warehouses. Define el término y trata los problemas principales involucrados en el desarrollo de un data warehouse. Su preocupación principal es la justificación del concepto y los asuntos operacionales y de diseño físico.

21.19 W. H. Inmon y R. D. Hackathorn: *Using the Data Warehouse.* Nueva York, N.Y.: Wiley (1994).

Es una explicación para usuarios y administradores del data warehouse. Al igual que otros libros sobre el tema, se concentra en asuntos físicos. Trata en detalle el concepto de almacén operational de datos.

21.20 P. G. W. Keen y M. S. Scott Morton: *Decision Support Systems: An Organizational Perspective.* Reading, Mass.: Addison-Wesley (1978).

Este texto clásico es uno de los primeros —si es que no *el* primero— en tratar explícitamente el apoyo para la toma de decisiones. La orientación es con relación al comportamiento y trata el análisis, diseño, implementación, evaluación y desarrollo de sistemas de apoyo para la toma de decisiones.

21.21 Ralph Kimball: *The Data Warehouse Toolkit.* Nueva York, N.Y.: John Wiley & Sons (1996).

Es un libro práctico. Como lo sugiere el subtítulo "Técnicas prácticas para la construcción de data warehouses dimensionales", el énfasis está centrado en los asuntos pragmáticos y no teóricos. La suposición tácita es que en esencia, no existe diferencia entre los niveles lógico y físico del sistema. Por supuesto, esta suposición es completamente válida para los productos actuales; sin embargo, sentimos que sería mejor tratar de mejorar la situación en lugar de simplemente aprobarla.

21.22 J. **D.** C. Little: "Models and Managers: The Concept of a Decision Calculus", *Management Science 16,* No. 8 (abril, 1970).

Este artículo presenta un sistema (Brandaid) diseñado para apoyar decisiones de productos, promociones, precios y publicidad. El autor identifica cuatro criterios para el diseño de modelos para apoyar la toma de decisiones de la administración: robustez, facilidad de control, simplicidad y compleción de los detalles relevantes.

21.23 M. S. Scott Morton: "Management Decision Systems: Computer-Based Support for Decision Making", Harvard University, Division of Research, Graduate School of Business Administration (1971).

Éste es el artículo clásico que presentó el concepto de los sistemas de decisiones para la administración, llevando claramente el apoyo para la toma de decisiones al campo de los sistemas basados en computadora. Se construyó un "sistema de decisiones para la administración" específico para coordinar la planeación de la producción para equipo de lavandería. Luego fue sometido a pruebas científicas con administradores de ventas y de producción como usuarios.

21.24 K. Parsaye y M. Chignell: *Intelligent Database Tools and Applications.* Nueva York, N.Y.: Wiley (1993).

Este libro parece ser el primero dedicado a los principios y técnicas de la minería de datos (aunque se refiere al tema como "bases de datos inteligentes").

21.25 A. Pirotte y P. Wodon: "A Comprehensive Formal Query Language for a Relational Data Base", *R.A.I.R.O. Informatique/Computer Science* ***11,*** No. 2 (1977).

21.26 R. **H.** Sprague y E. D. Carlson: *Building Effective Decision Support Systems.* Englewood Cliffs, N.J.: Prentice-Hall (1982).

Es otro texto clásico.

21.27 Erik Thomsen: *OLAP Solutions: Building Multi-Dimensional Information Systems.* Nueva York, N.Y.: Wiley (1997).

Es uno de los primeros libros sobre OLAP y tal vez el más completo. Se concentra en la comprensión de los conceptos y métodos de análisis usando sistemas multidimensionales. Es un intento serio para inyectar algo de disciplina en un tema confuso.

21.28 R. Uthurusamy: "From Data Mining to Knowledge Discovery: Current Challenges and Future Directions", en U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth y R. Uthurusamy (eds.): *Advances in Knowledge Discovery and Data Mining.* Cambridge, Mass.: AAAI Press/MIT Press (1996).

RESPUESTAS A EJERCICIOS SELECCIONADOS

21.8 Hay ocho ($= 2^3$) posibles agolpamientos para cada jerarquía y por lo tanto, la cantidad total de posibilidades es $8^4 = 4{,}096$. Como ejercicio adicional, tal vez quiera considerar lo que implica usar SQL para obtener todos estos resúmenes.

21.9 Con respecto a las consultas SQL mostramos solamente las cláusulas GROUP BY:

a. `GROUP BY GROUPING SETS ( (V#,P#), (P#,Y#), (Y#,V#) )`

b. `GROUP BY GROUPING SETS ( Y#, (Y#,P#), () )`

c. La trampa es que la consulta es ambigua; el término (por ejemplo) "enrollado en toda la dimen sión de proveedor" tiene muchos significados posibles. Sin embargo, una interpretación posible del requerimiento conducirá a una cláusula GROUP BY como ésta:

GROUP BY ROLLUP (V#), ROLUUP (P#)

d. GROUP BY CUBE (V#, P#)

Omitimos las tablas de resultados SQL. Respecto de las tabcruz, debe quedar claro que no son una forma muy buena para desplegar un resultado que involucra más de dos dimensiones (y entre más dimensiones hay, peor se pone). Por ejemplo, una de estas tabcruz —que corresponde a GROUP BY V#, P#, Y#— podría verse como esto (en parte):

	P1				P2				
	Y1	Y2	Y3		Y1	Y2	Y3		
V1	200	0	0		0	0	0		
V2	0	0	0		0	0	0		
V3	0	0	0		0	0	0		
V4	0	0	0		0	0	0		
V5	0	200	0		0	0	0		

En pocas palabras: los encabezados están amontonados y los arreglos son poco densos.

CAPITULO 22

Bases de datos temporales

22.1 INTRODUCCIÓN

Nota: Hugh Darwenfue el autor original de este capítulo.

En términos generales, una **base de datos temporal** es aquella que contiene datos históricos en vez (o además) de datos actuales. Tales bases de datos han sido investigadas desde mediados de los años setenta. Algunas de esas investigaciones adoptan la posición extrema de que los datos de dichas bases de datos sólo son insertados y nunca son eliminados ni actualizados (vea la explicación sobre los *data warehouses* en el capítulo anterior) y en ese caso, la base de datos contiene *solamente* datos históricos. El otro extremo es la base de datos de **instantánea*** que contiene solamente datos actuales, y los datos son eliminados o actualizados cuando los hechos representados por éstos dejan de ser ciertos (en otras palabras, una base de datos de instantánea es simplemente una base de datos como se entiende comúnmente y no una base de datos temporal).

A manera de ejemplo, considere nuevamente la base de datos de proveedores y partes de la figura 3.8. Esta base de datos es por supuesto una base de datos de instantánea y muestra entre otras cosas que el status del proveedor V1 es 20. Por el contrario, una versión temporal de esa base de datos podría mostrar no sólo que el status es actualmente 20, sino también que ha sido 20 desde el 1 de julio y que tal vez era 15 desde el 5 de abril hasta el 30 de junio, y así sucesivamente.

En una base de datos de instantánea, el tiempo de la instantánea es tomado generalmente como "ahora" (es decir, el momento en el cual se está inspeccionando en realidad a la base de datos). Aun cuando el tiempo de la instantánea sea un poco diferente a "ahora", no tiene diferencia real con respecto a la forma en que los datos son administrados y usados. Sin embargo, como veremos, la manera de administrar y usar los datos en una base de datos temporal difiere (en una variedad de formas importantes) de la manera de administrarlos y usarlos en una base de datos instantánea; de ahí la importancia del presente capítulo.

La característica distintiva de una base de datos temporal es, por supuesto, el tiempo mismo. Por lo tanto, la investigación sobre bases de datos temporales ha involucrado muchos análisis sobre la propia naturaleza del tiempo. Éstos son algunos de los aspectos que han sido estudiados:

- El aspecto filosófico de si el tiempo tiene inicio y fin;
- El aspecto científico de si el tiempo es continuo o si se presenta en cantidades discretas;
- El aspecto sicológico de cuál es la mejor manera de caracterizar el concepto importante de *ahora* (al que a menudo llamamos *"punto movible* ahora");

* Nada que ver con las instantáneas en el sentido que damos en el capítulo 9.

y así sucesivamente. Pero estas cuestiones (aunque sean muy interesantes por sí mismas) no son particularmente aspectos de las *bases de datos,* y por lo tanto en este capítulo no profundizamos en ellas; en cambio, sólo hacemos lo que esperamos sean suposiciones razonables en los lugares adecuados. Este enfoque nos permite concentrarnos en temas que son específicamente más importantes para nuestro propósito general. Sin embargo, observamos que parte de la investigación temporal ha conducido a generalizaciones interesantes que sugieren con frecuencia que las ideas desarrolladas para soportar los datos temporales también pueden tener aplicación en otras áreas. (Sin embargo, a pesar de este último punto, seguimos las convenciones al referirnos —a lo largo de este capítulo— a claves "temporales", operadores "temporales", relaciones "temporales" y así sucesivamente; aunque los conceptos en cuestión a menudo no sean exclusivos de los datos temporales como tales.)

¡Advertencia al lector! A continuación nos concentramos en lo que nos parece ser la más interesante e importante de las distintas ideas de investigación (en otras palabras, el capítulo es nuestro intento para extraer y explicar "las partes buenas" de esa investigación, aunque aquí nos apartamos de la literatura sobre aspectos de nomenclatura y otras cosas de menor importancia). Sin embargo, tenga presente que poca, si no es que nada, de la tecnología que describimos aquí ha aparecido en algunos DBMS comerciales. Las razones posibles para este hecho incluyen las siguientes:

- Tiene relativamente poco tiempo que el almacenamiento en disco ha llegado a ser lo sufi cientemente barato para que el almacenamiento de grandes volúmenes de datos históricos representen una proposición práctica. Como vimos en el capítulo 21, los "data warehouses" están llegando a ser ahora una realidad muy difundida; como consecuencia, los usuarios se encontrarán cada vez más con problemas de bases de datos temporales y comenzarán a bus car soluciones a esos problemas.
- Aunque la mayoría de las características que describimos (si no es que todas) han sido implementadas en forma de prototipos, su incorporación en productos existentes —en espe cial en productos SQL, donde es necesario porporcionar la separación de SQL con respecto al modelo relacional— podría ser un prospecto desalentador. Además, muchos de los fa bricantes están completamente ocupados intentando proporcionar soporte *objetos/relacional* (vea el capítulo 25).
- La comunidad de investigadores está todavía dividida sobre la mejor forma de atacar el problema (y esta falta de consenso puede haber llegado hasta los fabricantes). Algunos in vestigadores favorecen un enfoque muy especializado —uno que se aparta un poco de los principios relaciónales— que busca proveer específicamente datos temporales y que deja sin resolver otros problemas (vea por ejemplo la referencia [22.4]). Otros se inclinan hacia el suministro de operadores de propósito más general que puedan proporcionar (si así se desea) una base para el desarrollo de un enfoque especializado y al mismo tiempo no apar tarse del marco de referencia relacional (vea, por ejemplo, la referencia [22.3]). No hace falta decirlo, estamos a favor de este último enfoque.

Separamos una explicación de la estructura del capítulo para la sección que viene a continuación.

22.2 DATOS TEMPORALES

Si los datos son una representación codificada de los hechos, entonces los datos temporales son una representación codificada de hechos con **marcas de tiempo.** En una base de datos temporal (de acuerdo a la interpretación extrema de este término) *todos* los datos son tempo-

rales, lo que significa que cada hecho registrado tiene una marca de tiempo. De esto se desprende que una *relación temporal* es aquella en la cual cada tupia incluye al menos una marca de tiempo (es decir, el encabezado incluye al menos un atributo de algún tipo de marca de tiempo). Se desprende además que una *varrel temporal* es aquella cuyo encabezado es el de una relación temporal, y una *base de datos temporal* (relational) es aquella en la cual todas las varrels son temporales. *Nota:* Aquí somos deliberadamente ambiguos respecto de lo que pueda ser "algún tipo de marca de tiempo". Ahondaremos en este tema en las secciones 22.3 a 22.5.

Una vez proporcionada una definición razonablemente precisa del concepto "base de datos temporal" (en su forma extrema), ahora descartamos ese concepto, ¡ya que no es muy útil! Lo descartamos debido a que aunque las varrels originales en la base de datos sean todas temporales, muchas relaciones que pueden derivarse de esa base de datos (como los resultados de consultas) *no* son temporales. Por ejemplo, la respuesta a la consulta "obtener los nombres de todas las personas que hemos empleado alguna vez" puede ser obtenida a partir de alguna base de datos temporal, pero no es una relación temporal en sí misma. Y sena un poco extraño que un DBMS —en realidad, uno que no fuera relacional— nos permitiera obtener resultados que por sí mismos no pudieran ser conservados en la base de datos.

Por lo tanto, en este capítulo tomamos a una base de datos temporal como una base de datos que incluye algunos datos temporales, pero que no está limitada simplemente a los datos temporales. El resto del capítulo explica estas bases de datos en detalle. Entonces, el plan del capítulo es el siguiente:

- El resto de esta sección y la sección 22.3 preparan el escenario para las secciones subse cuentes; en particular, la sección 22.3 muestra por qué los datos temporales parecen requerir de un tratamiento especial.
- Las secciones 22.4 y 22.5 presentan a los *intervalos* como una forma conveniente para las marcas de tiempo de los datos. Luego, las secciones 22.6 y 22.7 explican una variedad de operadores escalares y de totales para manejar tales intervalos.
- La sección 22.8 presenta algunos operadores relaciónales nuevos que son importantes para operar sobre las relaciones temporales.
- La sección 22.9 analiza la cuestión de las restricciones de integridad para datos tempo rales. La sección 22.10 trata los problemas especiales de la actualización de tales datos.
- Por último, la sección 22.11 propone algunas ideas relevantes (y posiblemente novedosas) sobre el diseño de bases de datos y la sección 22.12 presenta un resumen.

Nota: Es importante comprender que —con una sola excepción, el *generador de tipo de intervalo* que se presenta en la sección 22.5— todos los operadores nuevos y otras construcciones que trataremos a continuación, son simplemente abreviaturas. Es decir, todos ellos pueden ser expresados (aunque a veces en forma muy larga) en términos de características que ya están disponibles en un lenguaje relacional completo, como **Tutorial D.** Justificaremos esto conforme avancemos (en algunos casos, pero no en todos).

Algunos conceptos y cuestiones básicas

Comenzamos apelando contra la forma en que la gente expresa lo que podríamos llamar "declaraciones con marca de tiempo" en lenguaje natural. Estos son tres ejemplos:

1. El proveedor VI fue nombrado (es decir, colocado bajo contrato) **en** el 1 de julio de 1999.
2. El proveedor VI ha sido un proveedor contratado **a partir** del 1 de julio de 1999.
3. El proveedor VI fue un proveedor contratado **durante** el periodo que va del 1 de julio de 1999 hasta el día actual.

Cada una de estas declaraciones es una interpretación posible de una 2-tupla que contiene el número de proveedor "VI" y la marca de tiempo "1 de julio de 1999", y cada una de éstas podría ser adecuada para esa 2-tupla si apareciera en una base de datos de instantánea que representara la situación actual en alguna empresa. Las preposiciones en negritas **en, a partir y durante** caracterizan las diferentes interpretaciones. *Nota:* A lo largo de este capítulo usamos "desde" y "durante" en los sentidos estrictos de **"después de" y "desde el principio hasta el fin"** (del periodo en cuestión), respectivamente, y diremos expresamente cuando no sea así.

Ahora, aunque nos acabamos de referir a *tres* posibles interpretaciones, podemos argumentar que las declaraciones 1, 2 y 3 en realidad están diciendo lo mismo en forma ligeramente diferente. De hecho, tomamos como equivalentes a las declaraciones 2 y 3, aunque no a la 1 y la 2 (ni la 1 y la 3). Consideramos:

- La declaración 1 establece claramente que VI no fue un proveedor contratado en la fecha (30 de junio de 1999) que precede a la fecha de nombramiento especificada; la declaración 2 ni establece ese hecho ni lo implica.
- Suponga que hoy ("el día actual") es el 25 de septiembre del 2000. Entonces la declaración 2 establece claramente que VI fue un proveedor contratado en cada uno de los días que van desde el 1 de julio de 1999 hasta el 25 de septiembre del 2000 (inclusive); la declaración 1 ni establece *ese* hecho ni lo implica.

Por lo tanto, las declaraciones 1 y 2 no son equivalentes y ninguna de ellas implica a la otra.

Una vez dicho esto, las tupias en una base de datos de instantánea con frecuencia incluyen cosas como "fecha de nombramiento", y las declaraciones como la 2 (o la 3) a menudo son la interpretación pretendida. Si éste es el caso aquí, la declaración 1 en su forma actual no es una interpretación completamente precisa de la tupia en cuestión. Podemos hacer que sea más precisa redactándola de esta forma: "el proveedor VI fue nombrado *más recientemente* en el 1 de julio de 1999". Lo que es más, si esta versión de la declaración 1 es en realidad lo que debe significar nuestra 2-tupla hipotética, entonces la declaración 2 en su forma actual no es tampoco una interpretación completamente precisa y necesita ser redactada de esta forma: "el proveedor VI no era un proveedor contratado al 30 de junio de 1999, pero lo ha sido a partir del 1 de julio de 1999".

Observe ahora que la declaración 1 expresa un tiempo **en** el cual se realizó determinado **evento** y en cambio, las declaraciones 2 y 3 expresan un intervalo de tiempo **durante** el cual persistió un **estado** determinado. Escogimos deliberadamente un ejemplo en el cual es posible inferir un *estado* determinado a partir de información que se refiere a un *evento* determinado; puesto que VI fue nombrado más recientemente en el 1 de julio de 1999, ese proveedor ha estado en el estado contratado desde esa fecha hasta el día actual. La tecnología de bases de datos clásica puede manejar razonablemente bien los *instantes de tiempo* (los tiempos en los cuales suceden los eventos), pero no maneja muy bien los *intervalos de tiempo* (periodos durante los cuales persisten los estados), como veremos en la sección 22.3.

Observe a continuación que aunque las declaraciones 2 y 3 son lógicamente equivalentes, su forma es significativamente diferente. Para ser más específicos, la forma de la declaración 2 no puede ser usada para registros históricos, mientras que la declaración 3 sí (siempre y cuando

la frase "el día actual" sea reemplazada en esa declaración con alguna fecha explícita, digamos 25 de septiembre del 2000). Por supuesto, la declaración correspondería entonces a una 3-tupla y no a una 2-tupla. Concluimos que el concepto "durante" es muy importante para registros históricos; al menos para los datos de estado aunque no para los datos de evento.*

Terminología: A los tiempos en los cuales sucedió determinado evento (o al intervalo durante el cual persistió determinado estado) se les conoce a veces como *tiempo válido.* Para ser más preciso, el **tiempo válido** de una proposición *p* es el conjunto de tiempos en los cuales se cree que/7 es *verdadera.* Es diferente al **tiempo de transacción,** el cual es el conjunto de tiempos en los que/? estuvo de hecho representada en la base de datos como *verdadera.* Los tiempos válidos pueden ser actualizados para reflejar el cambio de creencias, pero los tiempos de transacción no; es decir, los tiempos de transacción son mantenidos completamente por el sistema y no está permitido que los usuarios los cambien (por supuesto, están generalmente grabados, explícita o implícitamente, en la bitácora de transacciones).

Nota: Las referencias a *intervalos* y *conjuntos de tiempos* en el párrafo anterior, presentan tácitamente una idea simple, pero fundamental, de que un intervalo que tiene un tiempo inicial *s* y un tiempo de terminación *e* denota de hecho el conjunto de todos los tiempos *t* tales que $s < t < e$ (donde "<" significa "antes que", por supuesto). Aunque parece "obvia", esta noción simple tiene consecuencias de largo alcance, como veremos en las secciones siguientes.

Ahora bien, gran parte de la explicación anterior pretendió deliberadamente que usted se hiciera determinadas preguntas. Sin tomar en cuenta si tuvimos éxito en esto, ahora presentamos explícitamente estas preguntas y tratamos de responderlas.

1. ¿Acaso la expresión "todos los tiempos *t* tales que $s < t < e$" no presenta el espectro de los conjuntos infinitos y las dificultades conceptuales y computacionales que sufren estos conjuntos?

   Respuesta: Bueno, sí; eso parece. Pero descartamos el espectro y sorteamos las dificultades adoptando la suposición de que la "línea de tiempo" consiste en una secuencia finita de *quanta de tiempo* independientes e indivisibles. El intervalo con tiempo inicial s y tiempo final *e* involucra por lo tanto una cantidad finita de tales quanta, obligadamente.

   Nota: Gran parte de la literatura se refiere a un quantum de tiempo como *chronon.* Sin embargo, después continúan definiendo un chronon como un *intervalo* (vea por ejemplo, el glosario en la referencia [22.2]), lo que implica que tiene un punto inicial y otro final, y tal vez más puntos intermedios; por lo tanto, a fin de cuentas no es indivisible. (¿Qué son exactamente esos puntos? ¿Qué otra cosa pueden ser sino chronons?) Aquí encontramos algunas confusiones y por lo tanto decidimos evitar el término.

2. Las declaraciones 1, 2 y 3 parecen suponer que los quanta de tiempo son *días,* pero con seguridad el sistema soporta precisiones de tiempo de hasta pequeñas fracciones de segundo. Si VI fue un proveedor el 1 de julio de 1999 pero no el 30 de junio de 1999, ¿qué va hacer con el periodo que supuestamente hay entre el inicio del 1 de julio hasta el preciso instante del nombramiento, y durante el cual VI todavía no estaba oficialmente contratado?

   Respuesta: Necesitamos distinguir cuidadosamente entre los quanta de tiempo como tales, que son la unidad de tiempo más pequeña que el sistema puede representar, y las unidades de tiempo útiles para algún propósito en particular, y que pueden ser años, meses, días,

---

* Éste es un lugar tan bueno como cualquier otro para hacer notar que a pesar de nuestro repetido uso de términos como "registros históricos", las bases de datos temporales también pueden contener información que se refiere al futuro. Por ejemplo, tal vez queramos registrar el hecho de que el proveedor VI será un proveedor contratado durante el periodo desde *a* hasta *b* (donde *a* y *b* son fechas en el futuro).

semanas, etcétera. A estas unidades las llamamos **puntos de tiempo** (las abreviamos como *puntos)* para enfatizar el hecho de que *para lo que estamos haciendo,* también los consideramos indivisibles. Ahora podríamos decir *informalmente* que un punto de tiempo es "una sección en la línea del tiempo" —es decir, el conjunto de quanta de tiempo— que se alarga desde un quantum "límite" al siguiente (por ejemplo, desde la media noche de un día hasta la media noche del día siguiente). Por lo tanto, podríamos decir (de nuevo informalmente) que los puntos de tiempo tienen una duración, que en nuestro ejemplo es de un día. Sin embargo, *formalmente* los puntos de tiempo son (repitiendo) indivisibles y el concepto estricto de duración no puede ser aplicado.

Nota: Gran parte de la literatura usa el término *granulo* para referirse a algo parecido al punto de tiempo que acabamos de definir. Sin embargo, así como sucede con el término *chronon,* desafortunadamente después continúan diciendo que un granulo es un *intervalo.* Por lo tanto, también decidimos evitar el término *granulo** Sin embargo, sí utilizamos el término (informal) **granularidad,** la cual definimos (otra vez informalmente) como la duración del punto de tiempo aplicable. Por lo tanto, podemos decir que en nuestro ejemplo la granularidad es de un día, para indicar que dejamos a un lado —en este contexto— nuestra noción usual de que un día está compuesto de horas, que a su vez están compuestas de minutos, etcétera, (tales nociones pueden ser expresadas sólo recurriendo a niveles más finos de granularidad).

3. Tomando en cuenta entonces que la línea de tiempo es básicamente una secuencia de puntos de tiempo (de alguna granularidad), podemos referirnos sin confusión al "tiempo inmedia tamente subsiguiente" (o precedente) de cualquier punto dado. ¿Está correcto?

   Respuesta: Sí, hasta alcanzar un punto, siendo este punto, por supuesto, ¡el final del tiempo!; y en el otro extremo, el inicio del tiempo. Con respecto a lo que nos interesa, el inicio del tiempo es un punto de tiempo que no tiene predecesor (podría corresponder probablemente a la estimación más acertada de los cosmólogos para el preciso momento del supuesto Big Bang); el final del tiempo es un punto de tiempo que no tiene sucesor.

4. Si alguna relación incluye una 3-tupla que representa el hecho de que el proveedor VI es tuvo contratado desde el 1 de julio de 1999 hasta el 25 de septiembre del 2000, ¿acaso la suposición de mundo cerrado (vea el capítulo 5) no exige que la misma relación también incluya, por ejemplo, una 3-tupla que represente el hecho de que VI estuvo contratado des de el 2 de julio de 1999 hasta el 24 de septiembre del 2000, y una multitud de otras 3-tuplas que representen otras consecuencias triviales de la 3-tupla original?

   Respuesta: ¡Muy bien! Es claro que necesitamos un predicado más restrictivo que nuestra interpretación general de tales 3-tuplas: "el proveedor V* estuvo contratado en cada uno de los días que van desde la fecha *s* hasta la fecha *e,* pero no el día inmediatamente anterior a *s* ni el día inmediatamente posterior a *e".f* Esta interpretación más restrictiva, en su

---

* Nos parece que la confusión sobre si chronons y granulos son intervalos procede de una confusión entre la intuición y el formalismo. Una creencia intuitiva acerca de la manera en que funciona el mundo es una cosa; un *modelo* formal es algo completamente diferente. En particular, podemos *creer* que la línea de tiempo es continua e infinita, pero la modelamos —para efectos de cálculos en particular— como independiente y finita. *Nota:* Mientras estemos en este tema, debemos decir también que aunque el concepto de "quantum de tiempo" (o "chronon") es útil como base para la explicación del modelo formal a un nivel intuitivo, no es en sí mismo parte del modelo y no tiene nada que hacer ahí.

t A lo largo de este capítulo usamos el término no calificado "predicado" para referirnos a lo que en el capítulo 8 llamamos el predicado *externo* o entendido por el usuario, y no al predicado *interno* o entendido por el sistema (este último es, por supuesto, el predicado de *varreî).* Además, ignoramos aspectos de tales predicados externos que son "obvios" o que no están relacionados con el tema que estamos tratando.

forma general, proporciona la motivación y la base para muchos de los operadores que describimos en este capítulo; en particular en las secciones 22.8 y 22.10.

22.3 ¿CUÁL ES EL PROBLEMA?

En el resto de este capítulo continuaremos usando a los proveedores y partes como base para nuestros ejemplos, pero necesitamos hacer claramente algunas modificaciones a la base de datos para que sirva adecuadamente a nuestro propósito. Hacemos estas modificaciones poco a poco. En primer lugar eliminamos la varrel P de partes. Segundo, modificamos la varrel VP de envíos descartando el atributo CANT (y dejando solamente V# y P#) e interpretamos esa varrel VP modificada como: "El proveedor V# es *actualmente capaz* de proporcionar la parte P#" (en otras palabras, en vez de referirnos a los envíos *reales* de partes hechos por los proveedores, ahora la varrel se refiere solamente a envíos *potenciales;* es decir, a la *capacidad* de los proveedores para proporcionar partes). La figura 22.1 es una versión modificada de la figura 3.8 del capítulo 3, y muestra un conjunto de valores de ejemplo para esta base de datos modificada. Observe cuidadosamente que la base de datos sigue siendo instantánea; aún no incluye ningún aspecto temporal.

V#	PROVEEDOR	STATUS	CIUDAD
V1	Smith	20	Londres
V2	Jones	10	París
V3	Blake	30	París
V4	Clark	20	Londres
V5	Adams	30	Atenas

VP

V#	P#
V1	P1
V1	P2
V1	P3
V1	P4
V1	P5
V1	P6
V2	P1
V2	P2
V3	P2
V4	P2
V4	P4
V4	P5

Figura 22.1 La base de datos de proveedores y envíos (valores de ejemplo); versión de instantánea actual.

Ahora continuamos para tratar algunas restricciones y consultas simples para esta base de datos. Posteriormente consideraremos lo que les sucede a estas restricciones y consultas cuando la base de datos es extendida para incluir diversas características temporales.

Restricciones (base de datos de instantánea actual). Las únicas restricciones que queremos considerar son las diversas restricciones de *clave.* Sólo para recordarlo, {V#} y {V#, P#} son las claves primarias de V y VP respectivamente, y {V#} es una clave externa en VP que hace referencia a la clave primaria de V (ignoramos la clave externa {P#}, por supuesto).

Consultas (base de datos de instantánea actual). Consideramos solamente dos consultas, ambas muy simples:

- *Consulta 1.1:* Obtener los números de proveedor de los proveedores que pueden proporcionar actualmente alguna parte.

```
VP { V# }
```

■ *Consulta 1.2:* Obtener los números de proveedor de los proveedores que no pueden proporcionar actualmente ninguna parte.

```
V { V# } MINUS VP { V# }
```

Observe que la *consulta 1.1* involucra una **proyección** simple y la *consulta 1.2* involucra la **diferencia** entre dos de esas proyecciones. Posteriormente, cuando consideremos las analogías temporales de estas dos consultas, encontraremos que involucran analogías temporales de estos dos operadores (vea la sección 22.8). *Nota:* Probablemente no le sorprenderá aprender que también es posible definir equivalentes temporales de otros operadores relaciónales (vea el ejercicio 22.8).

Tratamiento "semitemporal" de proveedores y envíos

Para continuar, nuestro siguiente paso es tratar en forma "semitemporal" (por decirlo así) a las varrels V y VP añadiendo un atributo de marca de tiempo, A_PARTIR_DE, a cada una y renombrándolas adecuadamente. Vea la figura 22.2.

V_A_PARTIFLDE

V#	PROVEEDOR	STATUS	CIUDAD	A_PARTIR_DE
V1	Smith	20	Londres	*d04*
V2	Jones	10	París	*d07*
V3	Blake	319	París	*d03*
V4	Clark	20	Londres	*d04*
V5	Adams	30	Atenas	*002*

J>ARTIR_DE

V#	P#	A_PARTIR_DE
V1	P1	*d04*
V1	P2	*d05*
V1	P3	*d09*
V1	P4	*d05*
V1	P5	*d04*
V1	P6	*Ó06*
V2	P1	*d08*
V2	P2	*d09*
V3	P2	*d08*
V4	P2	*d06*
V4	P4	*CÍ04*
V4	P5	*d05*

Figura 22.2 La base de datos de proveedores y envíos (valores de ejemplo); versión semitemporal.

Por simplicidad, en la figura 22.2 no mostramos las marcas de tiempo auténticas; en su lugar, usamos símbolos de la forma *dOl, dO2,* etcétera, donde la *"d"* puede pronunciarse adecuadamente como "día", una convención a la cual nos adherimos a lo largo de este capítulo. (Por lo tanto, nuestros ejemplos hacen uso de puntos de tiempo que son específicamente *días.)* Suponemos que el día 1 precede inmediatamente al día 2, el día 2 precede inmediatamente al día 3 y así sucesivamente; también eliminamos los ceros iniciales en expresiones tales como "día 1" (como puede ver).

El predicado para V_A_PARTIR_DE es "el proveedor V# ha sido nombrado PROVEEDOR, ha tenido el status STATUS, ha estado ubicado en la ciudad CIUDAD y ha estado contratado a partir del día A_PARTIR_DE". El predicado para VP_A_PARTIR_DE es "el proveedor V# ha podido proporcionar la parte P# a partir del día A_PARTIR_DE".

Restricciones (base de datos semitemporal). Las claves primaria y externa para esta base de datos "semitemporal" son las mismas que antes. Sin embargo, necesitamos una restricción adicional

—que puede ser vista como un *aumento* a la restricción de clave externa de VP_A_PARTIR_DE hacia V_A_PARTIR_DE— para expresar el hecho de que ningún proveedor puede proporcionar parte alguna antes de estar contratado. En otras palabras, si la tupia *vp* en VP_A_PARTIR_DE hace referencia a la tupia v en V_A_PARTIR_DE, el valor de A_PARTIR_DE en *vp* no debe ser menor que el que está en v:

```
CONSTRAINT AUM_VP_TO_V_FK
   IS_EMPTY ( ( ( V_A_PARTIFLDE RENAME A_PARTIRJ3E AS VA ) JOIN
                ( VP_A_PARTIR_DE RENAME A_PARTIR_DE AS VPA ) )
              WHERE VPA < VA ) ;
```

Con este ejemplo comenzamos a vislumbrar el problema. Tomando una base de datos "semitemporal", como la de la figura 22.2, probablemente tendremos que establecer muchas restricciones de "clave externa aumentada" como ésta, y pronto comenzaremos a desear poder contar con alguna abreviatura adecuada para el propósito.

Consultas (base de datos semitemporal). Ahora consideramos las versiones "semitemporales" de las *consultas 1.1 y 1.2*.

- *Consulta 2.1:* Obtener los números de los proveedores que actualmente pueden proporcionar alguna parte, mostrando en cada caso la fecha desde la cual han podido hacerlo.

Si el proveedor *Vx* puede proporcionar actualmente varias partes, entonces Vx ha podido proporcionar *alguna* parte desde la fecha A_PARTIR_DE más temprana mostrada para *Vx* en VP_A_PARTIR_DE (por ejemplo, si *Vx* es VI, esa fecha A_PARTIR_DE más temprana es *dO4).* Por lo tanto:

```
SUMMARIZE VP PER VP { V# } ADD MIN ( A_PARTIR_DE ) AS A_PARTIR_DE
```

Resultado:

V#	A_PARTIR_DE
V1	*d04*
V2	*d08*
V3	*d08*
V4	*d04*

- *Consulta 2.2:* Obtener los números de proveedor que no pueden proporcionar actualmente cualquier parte en absoluto, mostrando en cada caso la fecha a partir de la cual no han podido hacerlo.

En nuestros datos de ejemplo, sólo hay un proveedor que no puede proporcionar actualmente parte alguna, el proveedor V5. Sin embargo, no podemos deducir la fecha a partir de la cual V5 ha estado contratado pero sin poder proporcionar alguna parte, debido a que no hay la información suficiente en la base de datos (la base de datos es todavía sólo "íem¡temporal"). Por ejemplo, suponga que *dlO* es el día actual. Entonces podría ser que V5 hubiera podido proporcionar al menos alguna parte desde hace algún tiempo como *dO2,* cuando V5 fue contratado por primera vez, hasta algún tiempo después algún *dO9;* o si nos vamos al otro extremo, podría ser que V5 nunca hubiera podido proporcionar en absoluto parte alguna.

Para tener alguna esperanza de responder la *consulta 2.2* debemos terminar el tratamiento "temporal" de nuestra base de datos, o al menos la parte VP de ella. Para ser más precisos, debemos mantener registros históricos en la base de datos que muestren cuáles proveedores pudieron proporcionar cuáles partes y en qué momento, tal como lo muestra la subsección que viene a continuación.

Tratamiento temporal completo de proveedores y envíos

La figura 22.3 muestra una versión completamente temporal de proveedores y envíos. Observe que los atributos A_PARTIR_DE se han convertido en atributos DESDE, y cada varrel ha adquirido un atributo adicional de marca de tiempo llamado HASTA. Los atributos DESDE y HASTA juntos expresan la noción de un intervalo de tiempo durante el cual algo es verdadero; por esa razón, reemplazamos A_PARTIR_DE por DESDE_HASTA en los nombres de varrel. Puesto que ahora estamos llevando registros históricos, hay más tupias en esta base de datos que las que había en sus predecesoras (como puede ver). Para ser más claros, vamos a dar por hecho que la fecha actual es *dlO* y por lo tanto, *dlO* aparece como el valor HASTA para cada tupia que se refiere al estado actual de las cosas. *Nota:* Tal vez se pregunte qué mecanismo podría ocasionar que todos esos *dlO* fueran reemplazados por *dll* cuando sea medianoche. Por desgracia, tenemos que dejar este asunto por el momento; regresaremos a él en la sección 22.11.

Observe que la base de datos temporal de la figura 22.3 incluye toda la información de la base semitemporal de la figura 22.2, junto con información histórica que se refiere a un periodo anterior (desde *dO2* hasta *dO4),* durante el cual el proveedor V2 estuvo contratado. El predicado para V_DESDE__HASTA es: "el proveedor V# fue nombrado PROVEEDOR, tuvo status STATUS, estuvo ubicado en la ciudad CIUDAD y estuvo contratado desde el día DESDE (y no en el día inmediatamente anterior a DESDE) hasta el día HASTA (y no en el día inmediatamente posterior a HASTA)". El predicado para VP_DESDE_HASTA es similar.

V_DESDE_HASTA

V#	PROVEEDOR	STATUS	CIUDAD	DESDE	HASTA
V1	Smith	20	Londres	*d04*	*d10*
V2	Jones	10	París	*d07*	*d10*
V2	Jones	10	París	*d02*	*d04*
V3	Blake	30	París	*d03*	*d10*
V4	Clark	20	Londres	*d04*	*dW*
V5	Adams	30	Atenas	*d02*	*d10*

VP_DESDE_HASTA

V#	P#	DESDE	HASTA
V1	P1	*d04*	*d10*
V1	P2	*d05*	*dW*
V1	P3	*d09*	*dw*
VI	P4	*d05*	*dW*
V1	P5	*d04*	*dw*
V1	P6	*d06*	*dW*
V2	P1	*d02*	*d04*
V2	P2	*d03*	*d03*
V2	P1	*d08*	*d10*
V2	P2	*d09*	*d10*
V3	P2	*d08*	*d10*
V4	P2	*d06*	*d09*
V4	P4	*d04*	*d08*
V4	P5	*d05*	*d10*

Figura 22.3 La base de datos de proveedores y envíos (valores de ejemplo); primera versión completamente temporal, usando marcas de tiempo.

Restricciones (primera base de datos temporal). En primer lugar necesitamos protegernos contra el absurdo de un par DESDE-HASTA en el cual el punto de tiempo de HASTA sea anterior al punto de tiempo de DESDE:

```
CONSTRAINT V_DESDE_HASTA_OK
   IS_EMPTY ( V_DESDE_HASTA WHERE HASTA < DESDE ) ;
CONSTRAINT VP_DESDE_HASTA_OK
   IS_EMPTY ( VP_DESDE_HASTA WHERE HASTA < DESDE ) ;
```

Luego observe, por el subrayado doble de la figura 22.3, que hemos incluido el atributo DESDE en la clave primaria para V_DESDE_HASTA y VP_DESDE_HASTA. Por ejemplo, la clave primaria de V_DESDE_HASTA obviamente no puede ser sólo {V#}, porque entonces no podríamos tener al mismo proveedor contratado por más de un periodo continuo. Una observación similar se aplica a VP_DESDE_HASTA. *Nota:* Podríamos haber usado los atributos HASTA en vez de los atributos DESDE; de hecho, V_DESDE_HASTA y VP_DESDE_HASTA tienen dos claves candidatas y son buenos ejemplos de varrels en las que no hay una razón obvia para escoger alguna de estas claves como "primaria" [8.13]. Hacemos estas selecciones simplemente para que queden bien definidas.

Sin embargo, estas claves primarias no capturan por sí mismas todas las restricciones que quisiéramos. Considere por ejemplo a la varrel V_DESDE_HASTA. Debe quedar claro que si hay una tupia para el proveedor *Wx* en esa varrel —con el valor / en DESDE y el valor *t* en HASTA—, entonces queremos que *no* haya una tupia para el proveedor *Wx* en esa varrel que indique que Vx estaba contratado en el día inmediatamente anterior a/o en el día inmediatamente posterior a *t.* Por ejemplo, considere al proveedor VI para el que sólo tenemos una tupia V_DESDE_HASTA con DESDE = *dO4* y HASTA = *dlO*. El simple hecho que (V#, DESDE) sea la clave primaria para esta varrel es claramente insuficiente para impedir la aparición de una tupia VI adicional "que se traslapa" con (digamos) DESDE = *d02* y HASTA = *dO6;* lo que indica, entre otras cosas, que VI *estuvo* contratado el día inmediatamente anterior a *dO4*. Queda claro que lo que nos gustaría es que estas dos tupias de VI se fundieran en una sola tupia con DESDE = *dO2* y HASTA = *dlO**

El hecho de que {V#, DESDE} sea la clave primaria de V_DESDE_HASTA también es insuficiente para impedir la aparición de una tupia VI "contigua" con (digamos) DESDE = *dO2* y HASTA = *dO3*, que indique de nuevo que VI estuvo contratado en el día inmediato anteriora *dO4*. Igual que antes, lo que nos gustaría es que las tupias se fundieran en una sola tupia.

Ésta es una restricción que prohibe dicho traslape y contigüidad:

```
CONSTRAINT AUM_V_DESDE_HASTA_PK
   IS_EMPTY ( ( ( V_DESDE_HASTA RENAME DESDE AS D1, HASTA AS H1 ) JOIN (
                V_DESDEJHASTA RENAME DESDE AS D2, HASTA AS H2 ) ) WHERE
                ( H1 > D2 AND H2 > D1 ) OR
                      ( D2 « H1+1 OR D1 = H2+1 ) ) ;
```

¡Esta expresión es muy complicada!, sin mencionar que nos hemos tomado la gran libertad de escribir (por ejemplo) "Hl + 1" para designar al sucesor inmediato del día indicado por Hl, un punto al que regresaremos en la sección 22.5. *Nota:* Si damos por hecho que esta restricción

---

* Observe que *no* fundir tales tupias ¡sería casi tan malo como permitir los duplicados! Los duplicados equivalen a "decir dos veces lo mismo". Y esas dos tupias para VI con intervalos de tiempo traslapados dicen efectivamente "dos veces lo mismo"; para ser más específicos, ambas dicen que VI estuvo contratado en los días 4, 5 y 6.

está establecida (y se hace cumplir, por supuesto), algunos escritores se referirán a la combinación de atributos {V#, DESDE, HASTA} como *clave candidata temporal* (de hecho, una clave *primaria* temporal). Sin embargo, el término no es muy bueno debido a que la clave candidata "temporal" no es en primer lugar ¡una clave *candidata]* de la varrel que la contiene. (Por el contrario, en la sección 22.9 encontraremos "claves candidatas temporales" que son genuinamente claves candidatas en el sentido clásico.)

Luego observe cuidadosamente que la combinación de atributos {V#, DESDE} en la varrel VP_DESDE_HASTA *no* es una clave externa de VP_DESDE_HASTA a V_DESDE_HASTA (aunque involucre a los mismos atributos de la clave primaria de V_DESDE_HASTA). Sin embargo, necesitamos aseguramos que si un determinado proveedor aparece en VP_DESDE_HASTA, entonces el mismo proveedor también debe aparecer en V_DESDE_HASTA:

```
CONSTRAINT AUM_VP_HASTA_V_FK_AGAIN1
   VP_DESDE_HASTA { V# } < V_DESDE_HASTA { V# } ;
```

(usamos "<" con el significado "es un subconjunto de".)

Pero la restricción AUM_VP_HASTA_V_FK_AGAIN1 no es suficiente por sí misma; también necesitamos asegurarnos que —aunque se hayan realizado todas las fusiones deseadas de tupias— si VP_DESDE_HASTA muestra a algún proveedor como capaz de proporcionar alguna parte durante algún intervalo de tiempo, entonces V_DESDE_HASTA muestra a ese mismo proveedor como contratado durante ese mismo intervalo de tiempo. Podemos intentar lo siguiente:

```
CONSTRAINT AUM_VP_HASTA_V_FK_AGAIN2 /* ¡Precaución — incorrecto! *l
   IS^EMPTY ( ( ( VJ)ESDE_HASTA RENAME DESDE AS VD, HASTA AS VH ) JOIN (
   VP_DESDE_HASTA RENAME DESDE AS VPD, HASTA AS VPH ) ) WHERE VPD < VD OR
   VPH > VH ) ;
```

Sin embargo, como lo indica el comentario, esta especificación es incorrecta. Para ver por qué, hagamos que V_DESDE_HASTA sea como se muestra en la figura 22.3 y que VP_DESDE_HASTA incluya una tupia para el proveedor V2 con (digamos) DESDE = *dO3* y HASTA = *dO4.* Tal arreglo es claramente consistente y aun así la restricción AUM_VP_HASTA_V_FK_AGAIN2, como está establecida, en realidad lo prohibe.

No trataremos de resolver este problema aquí, sino que lo dejaremos para una sección posterior (sección 22.9). Sin embargo, hacemos notar como asunto de terminología que si, como dijimos antes, la combinación de atributos {V#, DESDE, HASTA} en la varrel V_DESDE_HASTA es vista como una "clave candidata temporal", entonces la combinación de atributos {V#, DESDE, HASTA} en la varrel VP_DESDE_HASTA puede ser vista como "clave *externa* temporal" (aunque de hecho no es una clave externa como tal). De nuevo, vea la sección 22.9 para mayores detalles.

Consultas (primera base de datos temporal). Éstas son ahora las versiones completamente temporales de las *consultas 1.1 y 1.2:*

- *Consulta 3.1:* Obtener las tercias V#-DESDE-HASTA para los proveedores que hayan podido proporcionar alguna parte en algún tiempo; donde DESDE y HASTA juntos indican un periodo continuo máximo durante el cual el proveedor V# pudo de hecho proporcionar alguna parte. *Nota:* Aquí usamos el término "máximo" como una abreviatura conveniente para que signifique (en este caso) que el proveedor V# no pudo proporcionar parte alguna en el día inmediato anterior a DESDE o posterior a HASTA.

■ *Consulta 3.2:* Obtener las tercias V#-DESDE-HASTA para los proveedores que no hayan podido proporcionar en absoluto ninguna parte en algún tiempo; donde DESDE y HASTA juntos designan un periodo continuo máximo durante el cual el proveedor V# no pudo de hecho proporcionar parte alguna.

Bien, tal vez quiera hacer una pausa para convencerse de que, al igual que nosotros, ¡en realidad preferirá ni siquiera intentar estas consultas! Sin embargo, si lo intenta, se encontrará con que *pueden* ser expresadas (aunque en forma extremadamente laboriosa) y seguramente notará que necesitan algún tipo de abreviatura.

Por lo tanto, en pocas palabras, el problema de los datos temporales es que nos conducen hacia restricciones y consultas que son irracionalmente complejas de especificar; a menos que el sistema proporcione algunas abreviaturas bien diseñadas, lo cual (hasta donde sabemos) no hacen los DBMS comerciales actuales.

22.4 INTERVALOS

Ahora emprenderemos nuestro desarrollo de este conjunto adecuado de abreviaturas. El paso inicial y más fundamental es reconocer la necesidad de manejar a los intervalos como tales, por derecho propio, en vez de tener que tratarlos como pares de valores por separado, como hemos estado haciendo hasta este momento.

¿Qué es exactamente un intervalo? De acuerdo con la figura 22.3, el proveedor VI pudo proporcionar la parte Pl durante el intervalo que abarca desde el día 4 hasta el día 10. Pero, ¿qué significa "desde el día 4 hasta el día 10"? Queda claro que están incluidos los días 5,6,7,8 y 9; pero, ¿qué pasa con los puntos inicial y final, los días 4 y 10? Sucede que, habiendo dado algún intervalo específico, a veces queremos ver a los puntos inicial y final especificados como incluidos en el intervalo y a veces no. Si el intervalo desde el día 4 hasta el día 10 incluye al día 4, decimos que está **cerrado** con respecto a su punto inicial; en caso contrario, decimos que está **abierto** con respecto a ese punto. De manera similar, si incluye al día 10 decimos que está cerrado con respecto a su punto final y en caso contrario, decimos que está abierto con respecto a ese punto.

Por lo tanto, indicamos formalmente un intervalo mediante su punto inicial y su punto final (en ese orden), precedidos por un corchete de apertura o un paréntesis de apertura y seguidos por un corchete de cierre o un paréntesis de cierre. Los corchetes son usados cuando el intervalo es cerrado y los paréntesis cuando es abierto. Entonces, por ejemplo, hay cuatro formas distintas para indicar el intervalo específico que va desde el día 4 hasta el día 10 inclusive:

[d04,dW]
[d04,d11)
(d03,dW]
(d03,d11)

Nota: Tal vez piense que es extraño usar (por ejemplo, un corchete de apertura y un paréntesis de cierre; el hecho es que hay buenas razones para permitir los cuatro estilos. Además, el estilo llamado "cerrado-abierto" (corchete de apertura, paréntesis de cierre) es el más usado en la práctica.* Sin embargo, el estilo "cerrado-cerrado" (corchete de apertura, corchete de cierre) es con seguridad el más intuitivo, y será el que utilizaremos en las partes siguientes.

---

* Para ver por qué puede ser ventajoso el estilo cerrado-abierto, considere la operación de dividir el intervalo *[d04,d!0]* justo antes, digamos, del día *dO7*. El resultado son los intervalos inmediatamente adyacentes *[dO4,dO7) y [dO7,dlO]*.

Si tomamos en cuenta que los intervalos como *[d04,d!0]* son valores por derecho propio, tiene sentido combinar los atributos DESDE y HASTA de —digamos— V_DESDE_HASTA (vea la figura 22.3) en un solo atributo, DURANTE, cuyos valores son tomados de algún **tipo de intervalo** (vea la siguiente sección). Una ventaja inmediata de esta idea es que evita la necesidad de hacer la selección arbitraria sobre cuál de las dos claves candidatas {V#, DESDE} y {V#, HASTA ¡ debe ser la primaria. Otra es que también evita la necesidad de decidir si los intervalos DESDE-HASTA de la figura 22.3 deben ser interpretados como cerrados o abiertos con respecto a cada DESDE y HASTA; de hecho, *[dO4,dlO], [dO4,dll), [dO3,dlO]* y *(dO3,dll)* se convierten ahora en cuatro representaciones distintas posibles del mismo intervalo, y no tenemos necesidad de saber cuál es la representación real (en caso de haberla). Otra ventaja adicional es que las restricciones de varrel para "protegerse contra el absurdo de que aparezca un par DESDE-HASTA en el que el punto de tiempo HASTA preceda al punto de tiempo DESDE" (como lo dijimos en la sección 22.3) ya no son necesarias, debido a que la restricción "DESDE < HASTA" está implícita en la noción de un tipo de intervalo (en términos generales). Otra ventaja más es que ahora ya no hay necesidad de hablar de "claves temporales", que en realidad no son claves en el sentido clásico (vea la sección 22.9). También es posible simplificar otras restricciones (de nuevo, vea la sección 22.9).

La figura 22.4 muestra lo que pasa con nuestra base de datos de ejemplo si adoptamos este enfoque.

VJJURANTE

v#	PROVEEDOR	STATUS	CIUDAD	DURANTE
V1	Smith	20	Londres	*[d04,d10]*
V2	Jones	10	París	*[d07,d10]*
V2	Jones	10	París	*[d02,d04]*
V3	Blake	30	París	*Íd03,d10]*
V4	Clark	20	Londres	*[d04,d10]*
V5	Adams	30	Atenas	*[d02,d10]*

VP_DURANTE

v#	P#	DURANTE
VI	P1	*[d04,d10]*
V1	P2	*[d05,d10]*
V1	P3	*[d09,d10]*
V1	P4	*[d05,d10]*
V1	P5	*[d04,d10]*
V1	P6	*[d06,d10]*
V2	P1	*[d02,d04]*
V2	P2	*[d03,d03]*
V2	P1	*[d08,dW]*
V2	P2	*[d09,d10]*
V3	P2	*[d08,d10]*
V4	P2	*1 d06, d09]*
V4	P4	*[d04,d08]*
V4	P5	*[d05,d10]*

Figura 22.4 La base de datos de proveedores y envíos (valores de ejemplo); versión final completamente temporal, con intervalos.

22.5 TIPOS DE INTERVALO

Nuestra explicación de intervalos en la sección anterior fue en su mayoría intuitiva por naturaleza, y ahora necesitamos acercarnos al tema de manera más formal. En primer lugar, observe que la granularidad del valor del intervalo *[dO4,dlO]* es "días". Para ser más precisos, podríamos decir que es de *tipo DATE,* y mediante este término nos referimos a ese miembro específico de la familia usual de tipos de datos "fechahora" cuya precisión es "día" (a diferencia de —digamos— "hora" o "milisegundos" o "mes"). Esta observación nos permite especificar el tipo exacto del valor del intervalo *[dO4,dlff* de la manera siguiente:

- Primero y antes que nada (por supuesto) se trata de algún tipo de **intervalo;** este hecho es suficiente por sí mismo para determinar los *operadores* que son aplicables al valor del in tervalo en cuestión (así como, por ejemplo, decir que un valor *r* es de algún tipo de *re lación,* es suficiente para determinar los operadores —como JOIN— que son aplicables a ese valor *r).*
- Segundo, el valor del intervalo en cuestión es específicamente un intervalo de una fecha a otra y *este* hecho es suficiente para determinar el conjunto de *valores de intervalo* que cons tituyen al tipo de intervalo en cuestión.

El tipo específico de *[dO4,dlO]* es, por lo tanto, INTERVAL(DATE), donde:

a. INTERVAL es un **generador de tipo** (similar a RELATION en **Tutorial D** —vea el capí tulo 5— o "array" en los lenguajes de programación convencional) que nos permite definir una variedad de tipos de intervalo específicos (vea más adelante otros comentarios), y

b. DATE es el **tipo de punto** de este tipo de intervalo específico.

Observe cuidadosamente que, en general, el tipo de punto *TP* determina el tipo *y la precisión* de los puntos inicial y final —y de todos los puntos intermedios— de los valores de tipo INTERVALO/"). (En el caso del tipo DATE, por supuesto, la precisión está implícita.)

Nota: Dijimos en el capítulo 4 que la precisión no es parte del tipo aplicable sino que debe ser vista como una *restricción de integridad.* Si tomamos las declaraciones DECLARE X TIMESTAMP(3) y DECLARE Y TIMESTAMP(ó), por ejemplo, X y Y son del mismo tipo pero están sujetas a restricciones diferentes (X está restringida a mantener valores de milisegundos y Y está restringida a mantener valores de microsegundos). Por lo tanto, decir que TIMESTAMP(3) —o DATE— es un tipo de punto legal es (en términos estrictos) reunir dos conceptos que más valdría mantener separados. En su lugar, sería preferible definir dos tipos, TI y T2, con una representación TIMESTAMP posible pero con diferentes "restricciones de precisión", y luego decir que TI y T2 —y no, por ejemplo, TIMESTAMP(3) y TIMESTAMP(6)— son tipos de punto legales. Sin embargo, por simplicidad, en este capítulo seguimos el uso convencional y damos por hecho (en su mayoría) que la precisión es parte del tipo.

¿Qué propiedades debe poseer un tipo para que sea legal como tipo de punto? Bien, hemos visto que un intervalo está indicado por sus puntos inicial y final; también hemos visto que (al menos informalmente) un intervalo consiste en un conjunto de puntos. Si vamos a ser capaces de determinar todo el conjunto de puntos, a partir únicamente del punto *s* inicial y el punto *e* final,

primero debemos ser capaces de determinar el punto que está inmediatamente después (en algún ordenamiento acordado) del punto i. Al punto inmediatamente siguiente, lo llamamos el **sucesor** de *s,* y por simplicidad acordemos referirnos a él como s+1. Entonces, la función por la cual determinamos s+la partir de ***s,*** es la **función sucesora** para el tipo de punto (y precisión) en cuestión. Esa función sucesora debe ser definida para cada valor del tipo de punto, con excepción de aquel que ha sido designado como "último". (También habrá un punto designado como "primero", que no es sucesor de ninguno.)

Una vez establecido que *s+1* es el sucesor de *s,* debemos determinar entonces si 5+1 viene o no después de *e,* de acuerdo con el mismo ordenamiento acordado para el tipo de punto en cuestión. De no ser así, *s+1* sí es un punto en *[s,e],* y ahora debemos considerar el siguiente punto *s+2.* Continuando este proceso hasta que lleguemos al primer punto *s+n* que esté después de *e* descubriremos cada uno de los puntos de *[s,e].*

Observe que *s+n* es de hecho el sucesor de *e* (es decir, en realidad viene *inmediatamente* después de *e),* ahora podemos decir con seguridad que la única propiedad que un tipo *TP* debe tener para que sea legal como tipo de punto, es que debe definir una función sucesora para éste. Para abreviar, debe existir un **ordenamiento total** para los valores de *TP* (y podemos por lo tanto suponer que los operadores de comparación usuales —"<", ">", etcétera— están disponibles y definidos para todos los pares de valores *TP).*

Dicho sea de paso, seguramente ya habrá observado que no estamos hablando específicamente sobre datos temporales. Además, la mayor parte de lo que resta de este capítulo trata sobre los intervalos en general (en vez de intervalos de tiempo en particular) aunque consideraremos determinados asuntos temporales específicamente en la sección 22.11.

Entonces, ésta es (al fin) una definición precisa:

- Hagamos que *TP* sea un tipo de punto. Entonces un **intervalo (o valor de intervalo)** *i* de tipo INTERVAL(7P) es un valor escalar para el cual están definidos dos operadores escalares monádicos (START y END) y un operador diádico (IN), tales que:
  a. START(i) y END© regresan cada uno un valor de tipo *TP.*
  b. START© < END©.
  c. Sea *p* un valor de tipo *TP.* Entonces *p* IN *i* es *verdadero* si y sólo si tanto START(¡) < *p* como *p* < END(í) son *verdaderos.*

Observe en esta definición el llamado a la función sucesora definida para el tipo *TP.* Observe también que los puntos inicial y final constituyen una *representación posible,* en el sentido (por ejemplo) del capítulo 5, para valores de tipo INTER*AL(TP)* (y por lo tanto, en **Tutorial D** podríamos referirnos posiblemente a START y END como THE_START y THE_END, respectivamente). Observe por último que, por definición, los intervalos siempre son no vacíos (es decir, siempre hay, al menos, un punto "IN" en cualquier intervalo dado).

Observe muy cuidadosamente que un valor de tipo INTER VAL(rP) es un valor **escalar;** es decir, no tiene componentes visibles para el usuario. Es cierto que tiene una representación posible —de hecho, varias representaciones posibles, como vimos en la sección anterior— y esa representación posible tiene a su vez, componentes visibles para el usuario, aunque el valor de intervalo como tal no los tiene. Otra forma para decir lo mismo es decir que los intervalos están *encapsulados.*

22.6 OPERADORES ESCALARES SOBRE INTERVALOS

En esta sección definimos algunos operadores escalares útiles (la mayoría de ellos se explican por sí mismos) que se aplican a los valores de intervalo. Considere el tipo de intervalo INTERVAL^/[1]). Dejemos que/? sea un valor de tipo *TP*. Continuaremos usando la notación*p+l,p+2,* etcétera, para indicar al sucesor de *p,* al sucesor de *p+l y* así sucesivamente (un lenguaje real podría proporcionar algún tipo de operador NEXT). En forma similar, usaremos la notación*p-\, p-2,* etcétera, para indicar el valor cuyo sucesor es *p,* el valor cuyo sucesor es *p-*1 y así sucesivamente (un lenguaje real podría proporcionar algún tipo de operador PRIOR).

Dejemos que *pl* y *p2* sean valores de *TP*. Luego definimos *MAX(pl,p2)* para que regrese *p2* cuando *pl < p2* es *verdadero* y *pl* en caso contrario, así como *MlN(pl,p2)* para que regrese *pl* cuando *pl < p2* es *verdadero y p2* en caso contrario.

La notación que ya hemos estado usando servirá para los **selectores** de intervalo (al menos en contextos informales). Por ejemplo, las invocaciones de selector [3,5] y [3,6) ambas producen un valor de tipo INTERVAL(INTEGER) cuyos puntos contenidos son 3,4 y 5. (Un lenguaje real podría requerir, probablemente, una sintaxis más explícita como en, por ejemplo, INTERVAL ([3,5]).)

Sea ***il*** el intervalo *[si,el]* de tipo INTER*VAL{TP).* Como ya hemos visto, START(í7) regresa *si y* END(/7) regresa ***el;*** además definimos STOP(¡7) que regresa *el+\.* Sea también *i2* el intervalo *[s2,e2],* de tipo INTERVAL(rP). Entonces definimos los siguientes operadores de **comparación** de intervalo que son muy claros. *Nota: A* estos operadores se les conoce a menudo como *operadores de Alien,* ya que fueron propuestos por primera vez por Alien en la referencia [22.1]. Como ejercicio, tal vez quiera intentar el trazado de algunos esquemas simples para ilustrarlos.

- *il = i2* es *verdadero* si y sólo si *si = s2* y *el = e2* son ambos *verdaderos.*
- *il* BEFORE ¡2 es *verdadero* si y sólo si *el < s2* es *verdadero.*
- *il* MEETS ¡2 es *verdadero* si y sólo si *s2 = el+* es *verdadero o si = e2+* es *verdadero.*
- *il* OVERLAPS *i2* es *verdadero* si y sólo si *si <e2 y s2 < el* son ambos *verdaderos.*
- ***il*** DURING *i2* es *verdadero* si y sólo si *s2 <sl* y *e2> el* son ambos *verdaderos**
- ***il*** STARTS *i2* es *verdadero* si y sólo si *si = s2 y el < e2* son ambos *verdaderos.*
- ***il*** FINISHES ¡2 es *verdadero* si y sólo si *el = e2 y si > s2* son ambos *verdaderos.*

Nota: Las definiciones de estos operadores también pueden ser dadas en términos *de puntos* (del tipo de punto aplicable). Por ejemplo, podríamos decir que *il* OVERLAPS *i2* es *verdadero* si y sólo si existe un valor/» de tipo *TP* tal que *p* IN *il* y *p* IN *i2* ambos son *verdaderos.*

Siguiendo la referencia [22.3], también podemos definir las siguientes adiciones útiles a los operadores de Alien:

- *il* MERGES *i2* es *verdadero* si y sólo si *il* MEETS *i2* es *verdadero* o *il* OVERLAPS *i2* es *verdadero.*
- *il* CONTAINS *i2* es *verdadero* si y sólo si *i2* DURING ***il*** es *verdadero?*

---

* Observe que aquí (sólo una vez) DURING no significa "a lo largo del intervalo en cuestión".

[f] Aquí, INCLUDES podría ser una mejor palabra clave que CONTAINS; entonces podríamos usar CONTAINS como el inverso de IN y definir a *i* CONTAINS *p* como equivalente de *p* IN *i.*

Para obtener la longitud, por decirlo así, de un intervalo tenemos DURATION^), que regresa la cantidad de puntos que hay en *i.* Por ejemplo, *DURATlON([d03,dO7])* = 5.

Por último, definimos algunos operadores diádicos útiles sobre intervalos que regresan intervalos:

- *il* UNION *i2* produce [MIN(í7,J2), *MAX(el,e2)]* cuando *il* MERGES *i2* es *verdadero* y en caso contrario, queda indefinido.
- ***il*** INTERSECT *i2* produce [MAX(íi,í2),MIN(ei,<?2)] cuando *il* OVERLAPS *i2* es *ver dadero* y en caso contrario, queda indefinido.

Nota: Aquí UNION e INTERSECT son los operadores generales de conjuntos y no sus contrapartes relaciónales especiales. La referencia [22.3] los llama MERGE e INTERVSECT, respectivamente.

22.7 OPERADORES DE TOTALES SOBRE INTERVALOS

En esta sección presentamos dos operadores extremadamente importantes, *UNFOLD* y *COALESCE.* Cada uno de estos operadores toma un conjunto de intervalos del mismo tipo como su único operando y regresa otro conjunto similar. El resultado en ambos casos puede ser considerado como una *forma canónica* particular del conjunto original (si necesita recordar a qué se refiere el término *forma canónica,* consulte el capítulo 17, sección 17.3).

La explicación que damos a continuación está motivada por observaciones como la siguiente. Sean *XI* y *X2* los conjuntos

```
{ [d01,d01], [d03,d05], [d04,d06] }
```

```
{ [d01,d01], [d03,d04], [d05,d05], [d05,d06] }
```

(respectivamente). Es fácil ver que *XI* no es el mismo conjunto que *X2.* También es fácil ver que (a) el conjunto de todos los puntos *p* tales que *p* esté contenido en algún intervalo en *XI* es el mismo que (b) el conjunto de todos los puntos/? tales que*p* esté contenido en algún intervalo en *X2* (los puntos en cuestión son *dOl, dO3, dO4, dO5* y *dO6).* Sin embargo, por razones que pronto aclararemos, estamos interesados no tanto en ese conjunto de puntos como tal, sino en el conjunto de **intervalos unitarios** correspondiente (llamémoslos *X3):*

```
{ [d01,d01], [d03,d03], [d04,d04], [d05,d05], [d06,d06] }
```

Decimos que *X3* es *Informa desdoblada* de *XI* (y *X2).* En general, si *X* es un conjunto de intervalos del mismo tipo, entonces la **forma desdoblada** de *X* es el conjunto de todos los intervalos de la forma *[p,p],* en donde/? es un punto que está en algún intervalo en *X.*

Observe que *Xl,X2y X3* difieren en cardinalidad. Ahora bien, en nuestro ejemplo sucede que *X3* (la forma desdoblada) es la que tiene mayor cardinalidad, pero es fácil encontrar un conjunto *X4* que tenga la misma forma desdoblada que *XI* y tenga cardinalidad mayor a la de *X3* (el ejercicio queda para el lector). También es fácil encontrar el mucho más interesante —y necesariamente único— conjunto *X5* que tenga la misma forma desdoblada y la cardinalidad *mínima posible:*

Id01,d01], [d03,d06]

Decimos que *X5* es la **forma fundida** de *XI* (y también de *X2, X3* y *X4).* En general, si *X* es un conjunto de intervalos —todos del mismo tipo— entonces la **forma fundida** de Xes el conjunto *Y* de intervalos del mismo tipo, tales que (a)IyF tienen la misma forma desdoblada y (b) no hay dos miembros distintos, ***il*** e *i2,* de *Y* tales que *il* MERGES *i2* sea *verdadero.* Observe que (como ya hemos visto) muchos conjuntos distintos pueden tener la misma forma fundida. Observe también que la definición *de. forma fundida* se apoya —aunque la definición *de. forma desdoblada* no lo hace— en la definición de función sucesora para el tipo de punto subyacente.

Ahora podemos definir los operadores **UNFOLD** y **COALESCE.** Sea *X* un conjunto de intervalos de tipo INTERVAL(7P). Entonces, UNFOLD(X) regresa la forma desdoblada de *X,* mientras que COALESCE(X) regresa la forma fundida de *X. Nota:* Debemos decir que *forma desdoblada y forma fundida* no son términos estándar; de hecho, no parece haber ningún término estándar para estos conceptos, aunque los conceptos como tales, estén ciertamente tratados en la literatura.

Estas dos formas canónicas juegan un papel importante en las soluciones a las que al fin estamos comenzando a aproximarnos para los problemas que tratamos en la sección 22.3. Sin embargo, los operadores UNFOLD y COALESCE todavía no son lo que necesitamos (siguen siendo tan sólo un paso en el camino). En vez de ello, lo que necesitamos son determinadas *contrapartes relaciónales* de esos operadores y definiremos dichas contrapartes en la sección que viene a continuación.

22.8 OPERADORES RELACIÓNALES QUE INVOLUCRAN INTERVALOS

Los operadores escalares sobre intervalos que describimos en la sección 22.6, están por supuesto disponibles para ser utilizados con expresiones escalares en los lugares usuales dentro de las expresiones relaciónales. Por ejemplo, en **Tutorial D** esos lugares son básicamente cláusulas WHERE en restricciones y cláusulas ADD en EXTEND y SUMMARIZE. Por lo tanto, si utilizamos la base de datos de la figura 22.4, la consulta "obtener los números de proveedor para los proveedores que pudieron proporcionar la parte P2 en el día 8" podría ser expresada de la siguiente forma:

```
( VPJURANTE WHERE P# • P# ('P2') AND d08 IN DURANTE ) {V#}
```

Nota: En la práctica, la expresión *"dO8"* que aparece aquí tendría que ser reemplazada por una literal adecuada de tipo DAY.

Como otro ejemplo, la siguiente expresión produce una relación que muestra qué pares de proveedores estaban ubicados en la misma ciudad al mismo tiempo, junto con las ciudades y tiempos en cuestión:

```
EXTEND
   ( ( ( ( V_DURANTE RENAME V# AS XV#,
                          DURANTE AS XD ) { XV#, CIUDAD, XD } JOIN
         ( VJURANTE RENAME V# AS YV#,
                       DURANTE AS YD ) { YV#, CIUDAD, YD } ) WHERE XD
OVERLAPS YD ) ADD ( XD INTERSECT YD ) AS DURANTE ) { XV#, YV#,
CIUDAD, DURANTE }
```

Explicación: JOIN encuentra partes de proveedores ubicados en la misma ciudad. WHERE restringe ese resultado a los pares que estuvieron en la misma ciudad al mismo tiempo. EXTEND ... ADD calcula los intervalos importantes. La proyección final da el resultado deseado.

Ahora regresamos a las *consultas 3.1 y 3.2* que se encuentran al final de la sección 22.3. Nos concentramos primero en la *Consulta 3.1*. La *Consulta 4.1* es un replanteamiento de esa consulta en términos de la base de datos de la figura 22.4:

- *Consulta 4.1:* Obtener los pares V#-DURANTE para los proveedores que hayan podido proporcionar alguna parte en algún tiempo, donde DURANTE designa un periodo continuo máximo durante el cual el proveedor V# en realidad pudo proporcionar alguna parte.

Recordará que una versión anterior de esta consulta, la *consulta 2.1*, requería el uso de a»rupamiento y agregación (más específicamente, involucraba una operación SUMMARIZE). Por lo tanto, probablemente no le sorprenderá saber que la *consulta 4.1* también requerirá determinadas operaciones de naturaleza agrupante y de agregación. Sin embargo, formularemos la consulta en pasos pequeños. El primer paso es:

```
WITH VP_DURANTE { V#, DURANTE } AS T1 :
```

(agregaremos más elementos a esta expresión, como lo sugieren los dos puntos). Este paso simplemente descarta números de parte. Por lo tanto, TI se ve de esta forma:

V#	DURANTE
V1	*[d04,d10]*
V1	*[d05,d10]*
V1	*[d09,d10]*
V1	*[d06,dW]*
V2	*[d02,d04]*
V2	*[d03,d03]*
V2	*[d08,d10]*
V2	*[d09,d10]*
V3	*[d08,d10]*
V4	*[d06, d09]*
V4	*[d04,d08]*
V4	*[d05,d10]*

Observe que esta relación contiene información redundante; por ejemplo, nos dice (no menos de tres veces) que el proveedor VI pudo proporcionar algo el día 6. El resultado deseado, si eliminamos todas estas redundancias, es claramente el siguiente (llamémoslo RESULT):

V#	DURANTE
V1V2	*[d04,d10]*
V2	*Id02,d04]*
V3	*[d08,d10]*
V4	*[d08,dW]*
	[d04,dW]

Llamamos a este resultado la **forma fundida** de TI **en** DURANTE. Observe cuidadosamente que en esta forma fundida, un valor DURANTE para un proveedor dado no existe necesariamente como un valor DURANTE explícito para ese proveedor en la relación TI a partir de

la cual la forma fundida está derivada (y en nuestro ejemplo esta observación se aplica en particular al proveedor V4).

Ahora bien, con el tiempo llegaremos a un punto donde podamos obtener esta forma fundida por medio de una simple expresión de la forma:

```
T1 COALESCE DURANTE
```

Sin embargo, necesitamos llegar gradualmente a ese punto.

Observe primero que hemos estado usando el término "forma fundida" en los dos párrafos anteriores en un sentido ligeramente diferente al que usamos en la sección 22.7. El operador COALESCE, tal como lo definimos en la sección 22.7, toma como entrada un conjunto de intervalos y produce como salida un conjunto de intervalos. Sin embargo, aquí estamos hablando de una versión diferente —de hecho, una *sobrecarga* (vea el capítulo 19)— de ese operador que toma una *relación uñaría* como entrada y produce otra relación igual (con el mismo encabezado) como salida, y son las tupias de esas relaciones las que contienen los intervalos reales.

Entonces, éstos son los pasos que nos llevan desde TI hasta RESULT:

```
WITH ( T1 GROUP ( DURANTE ) AS X ) AS T2 :
```

(si necesita refrescar su memoria con respecto al operador GROUP, consulte el capítulo 6). T2 luce de esta forma:

V#	X
V1	DURANTE *[d04,d10* *[d05,d10]* *[d09,d10]* *[d06,d10]*
V2	DURANTE *[d02,d04]* *[d03,d03]* *[d08,d10]* *Id09,d10]*
V3	DURANTE *[d08,d10]*
V4	DURANTE [*d06,d09]* *[d04,d08]* *[d05,d10]*

Ahora aplicamos la nueva versión de COALESCE a las relaciones que son valores del atributo X:

```
WITH ( EXTEND T2 ADD COALESCE ( X ) AS Y )
                                { ALL BUT X } AS T3
```

T3 se ve de esta forma:

V#	Y
V1 V2	DURANTE *[d04,d10]*
V3	DURANTE *[d02,d04]* *[d08,d10]*
V4	DURANTE *[d08,d10]*
	DURANTE *[d04,d10]*

Por último, desagrupamos (de nuevo, vea el capítulo 6 si es necesario):

```
T3 UNGROUP Y
```

Esta expresión produce la relación que llamamos anteriormente RESULT. En otras palabras, ahora mostramos todos los pasos juntos (y simplificamos ligeramente) y RESULT es el resultado de evaluar la siguiente expresión general:

```
WITH VP.DURANTE { V#, DURANTE } AS T1,
     ( T1 GROUP ( DURANTE ) AS X ) AS T2,
     ( EXTEND T2 ADD COALESCE ( X ) AS Y ) { ALL BUT X } AS T3 :
T3 UNGROUP Y
```

Obviamente, sería bueno poder ir desde TI hasta RESULT en una sola operación. Para ese fin, inventamos un nuevo operador "fundir relación" con la siguiente sintaxis:

```
fl COALESCE A
```

(donde *R* es una expresión relacional y A es un atributo —de algún tipo de intervalo— de la relación indicada por esa expresión).* La semántica de este operador está definida por la generalización obvia de las operaciones de agrupamiento, extensión, proyección y desagrupamiento

* Si fuera necesario, el operando A podría ser extendido para permitir una lista de nombres de atributo separados con comas. Un comentario similar se aplica también al operador "desdoblar relación" (vea más adelante). Para la semántica vea el ejercicio 22.8.

mediante las cuales obtenemos RESULT a partir de TI. *Nota:* Tal vez le ayude observar que fundir *R* sobre *A* involucra el agrupamiento de *R* por todos los atributos de *R* que no sean *A* (recuerde del capítulo 6 que —por ejemplo— la expresión "TI GROUP (DURANTE)..." puede ser leída como "agrupar TI por V#"; donde V# es el único atributo de TI que *no* es mencionado en la cláusula GROUP).

Si reunimos todo lo anterior, podemos ofrecer ahora lo siguiente como una formulación zonablemente directa de la *consulta 4.1:*

```
VP_DURANTE { V#, DURANTE } COALESCE DURANTE
```

La operación general indicada por esta expresión es un ejemplo de lo que algunos escritores llaman **proyección temporal.** Para ser más específicos, es una "proyección temporal" de **VPJ5URANTE** sobre V# y DURANTE. (Recuerde que la versión original de esta consulta, la *consulta 1.1,* involucra la proyección ordinaria de VP sobre V#.) Observe que la proyección temporal no es exactamente una proyección como tal, sino que es una "analogía temporal" de una proyección ordinaria.

Ahora pasemos a la *consulta 3.2.* La *consulta 4.2* es un replanteamiento de esta consulta en términos de la base de datos de la figura 22.4:

- *Consulta 4.2:* Obtener los pares V#-DURANTE para los proveedores que nó han podido proporcionar cualquier parte en algún tiempo, donde DURANTE designa a un periodo continuo máximo durante el cual el proveedor V# no pudo proporcionar parte alguna.

Recuerde que la versión original de esta consulta, la *consulta 1.2,* involucró una operación de diferencia relacional. Por lo tanto, si está esperando ver algo como una diferencia *temporal,* tiene usted razón. De igual forma, como era de esperar, así como la "proyección temporal" implica "fundir la relación", la "diferencia temporal" implica "desdoblar la relación".

La "diferencia temporal" (al igual que la operación de diferencia ordinaria) involucra dos operandos de relación. Nos concentraremos primero en el operando izquierdo. Si desdoblamos el resultado de la proyección (normal) V_DURANTE {V#, DURANTE} sobre DURANTE obtenemos una relación, llamémosla TI, que se ve de esta forma:

V#	DURANTE
V1	*[d04,d04]*
V1	*[d05, d05]*
V1	*í d06, d06]*
V1	*[d07, d07)*
V1	*[d08,d08]*
V1	*[d09, d09]*
V1	*[d10,dW*
V2	*I d07, d07)*
V2	*[d08, d08]*
V2	*[d09,d09]*
V2	*Id10,d10]*
V2	*[d02, d02)*
V2	*[d03, d03]*
V2	*[d04,d04]*
V3	*[d03, d03]*

Dados los datos de ejemplo de la figura 22.4, TI contiene de hecho un total de 23 tupias. (Ejercicio: verifique esta afirmación.)

Si definimos una versión de "relación uñaría" de UNFOLD (similar a la versión de "relación uñaría" de COALESCE), entonces podemos obtener TI de la siguiente forma:

```
( EXTEND ( V_DURANTE { V#, DURANTE } GROUP ( DURANTE ) AS X ) ADD
  UNFOLD ( X ) AS Y ) { ALL BUT X } UNGROUP Y
```

Sin embargo, como ya sugerimos, podemos simplificar las cosas inventando un operador "desdoblar relación" con una sintaxis como la siguiente (y semántica directa):

```
R UNFOLD A
```

Ahora podemos escribir

```
WITH ( V_DURANTE { V#, DURANTE } UNFOLD DURANTE ) AS T1 :
```

Tratamos de la misma forma el operando derecho de "diferencia temporal":

```
WITH ( VP_DURANTE { V#, DURANTE } UNFOLD DURANTE ) AS T2 :
```

Ahora podemos aplicar la diferencia de relación (normal):

```
WITH ( T1 MINUS T2 ) AS T3 :
```

T3 luce de esta forma:

V#	DURANTE
V2	[*d07, d07]*
V3	*Id03,d03]*
V3	*[d04,d04]*
V3	*[d05,d05]*
V3	*1 d06,d06]*
V3	*[d07,d07]*
V5	*[d02, d02]*
V5	*[d03,d03]*
V5	*[d04,d04]*
V5	*[d05, d05]*
V5	*Íd06,d06]*
V5	*1 d07, d07]*
V5	*[d08,d08]*
V5	*[d09,d09]*
V5	*[d10,d10]*

Por último, fundimos T3 en DURANTE para obtener el resultado deseado:

```
T3 COALESCE DURANTE
```

El resultado luce de esta forma:

V#	DURANTE
V2	[*d07, d07]*
V3	*[d03,d07]*
V5	*[d02,d10]*

Entonces, ésta es una formulación de la *consulta 4.2* como una sola expresión anidada:

```
( ( V_DURANTE { V#, DURANTE } UNFOLD DURANTE )
  MINUS
  ( VPJDURANTE { V#, DURANTE } UNFOLD DURANTE ) )
COALESCE DURANTE
```

Como ya indicamos, la operación general indicada por esta expresión es un ejemplo de lo que algunos escritores llaman **diferencia temporal.** De manera más precisa, es una "diferencia temporal" entre las proyecciones de V_DURANTE y VP_DURANTE (en ese orden) sobre V# y DURANTE. *Nota:* Al igual que la proyección temporal, la diferencia temporal no es exactamente una diferencia temporal, sino que es una "analogía temporal" de una diferencia ordinaria.

Sin embargo, aún no terminamos. Las expresiones de "diferencia temporal" como la que muestra el ejemplo, son requeridas tan frecuentemente en la práctica que parece que vale la pena definir una abreviatura adicional para ellas. * Para ser específicos, parece ser que vale la pena lograr, como si fuera una sola operación, la secuencia (a) desdoblar ambos operandos, (b) tomar la diferencia y luego (c) fundir. Además, como ganancia, tal abreviatura brinda la oportunidad de *mejorar el rendimiento.* Cuando están involucrados intervalos largos de granularidad fina, el resultado del desdoblamiento de una relación puede ser muy grande en comparación con sus operandos; de hecho, si el sistema materializa ambos desdoblamientos, calcula la diferencia y luego funde el resultado, tales consultas pueden seguir en ejecución "indefinida" o quedarse sin espacio en disco. Expresar la diferencia temporal como una sola operación podría ayudar a que el optimizador supiera lo que va a suceder y tal vez en realidad evite hacer cualquier desdoblamiento. Entonces, ésta es nuestra abreviatura adicional propuesta:

```
R1 I.MINUS R2 ON A
```

Aquí *Rl* y *R2* son expresiones relaciónales que denotan las relaciones *rl* y *r2* del mismo tipo, y *A* es un atributo de algún tipo de intervalo que es común para esas dos relaciones (y por supuesto, el prefijo "I_" significa "intervalo"). Como ya hemos visto (más o menos), esta expresión está definida para que sea semánticamente equivalente a lo siguiente:

```
( ( R1 UNFOLD A ) MINUS ( R2 UNFOLD A ) ) COALESCE A
```

Vea el ejercicio 22.2 para una explicación adicional de operadores "I_" como IJVHNUS.

22.9 RESTRICCIONES QUE INVOLUCRAN INTERVALOS

Está claro que la combinación de atributos {V#, DURANTE} es una clave candidata para la varrel V_DURANTE; de hecho, en la figura 22.4 usamos nuestra convención de doble subrayado para mostrarla específicamente como clave *primaria.* (Observe que {V#} por sí misma no es una clave candidata, debido a que es posible que el contrato de un proveedor termine y luego se le vuelva a contratar en una fecha posterior; vea por ejemplo, el proveedor V2 en la figura 22.4.) Por lo tanto, la varrel V_DURANTE puede ser definida de la siguiente forma:

```
VAR V_DURANTE BASE RELATION
  { V# V#,  PROVEEDOR NOMBRE,  STATUS INTEGER,  CIUDAD CHAR,
    DURANTE  INTERVAL ( DATE ) }
  KEY { V#,  DURANTE }  ;                    /* Advertencia - i Inadecuado I */
```

---

* Observe que (por el contrario) *no* definimos una abreviatura explícita para la proyección temporal.

Sin embargo, la especificación KEY, como la mostramos aquí (aunque *es* lógicamente correcta), también es inadecuada en cierto sentido, ya que falla en impedir que la varrel V_DURANTE contenga por ejemplo, las dos tupias siguientes:

V2	Jones	10	París	[*d02,d08]*
V2	Jones	10	París	*[d07,d10]*

Como puede ver, estas dos tupias muestran una cierta *redundancia,* ya que la información que se refiere al proveedor V2 para los días 7 y 8 está registrada dos veces.

La especificación KEY también es inadecuada de otra forma. Para ser más específicos, falla en impedir que la varrel V_DURANTE contenga, por ejemplo, las dos tupias siguientes:

V2	Jones	10	París	[*d02,d06]*
V2	Jones	10	París	*[d07,d10]*

Aquí no hay redundancia, pero hay cierta *ambigüedad,* ya que estamos utilizando dos tupias para decir lo que podríamos decir con una:

V2	Jones	10	París	*[d02,d10]*

Debe quedar claro que para impedir tales redundancias y ambigüedades, lo que necesitamos es hacer cumplir una restricción de varrel, llamémosla *restricción Cl,* que diga lo siguiente:

> Si dos tupias V_DURANTE distintas son idénticas, con excepción de sus valores DURANTE *il* e *i2,* entonces *il* MERGES *i2* debe *ser falso.*

(Recuerde que MERGES es el OR de OVERLAPS y MEETS, en términos generales; el reemplazo de MERGES por OVERLAPS en la restricción C1 nos da la restricción que necesitamos para hacer que se impida la redundancia, y el reemplazo por MEETS da la restricción que necesitamos para hacer que se impida la ambigüedad.) También debe quedar claro que hay una forma muy simple para hacer cumplir la restricción C1: manteniendo la varrel V_DURANTE **fundida** todo el tiempo sobre el atributo DURANTE. Por lo tanto, definamos una nueva cláusula COALESCED que puede aparecer opcionalmente en una definición de varrel como ésta:

```
VAR V_DURANTE BASE RELATION
    { V# V#, PROVEEDOR NOMBRE, STATUS INTEGER, CIUDAD CHAR,
      DURANTE INTERVAL ( DATE ) }
    KEY { V#, DURANTE }      /* Advertencia - ¡todavía inadecuado! */
    COALESCED DURANTE ;
```

En este caso, la especificación COALESCED DURANTE significa que la varrel V_DURANTE debe ser en todo momento idéntica al resultado de la expresión V_DURANTE COALESCE DURANTE (lo que implica que la fundición de V_DURANTE sobre DURANTE nunca tiene ningún efecto). Por lo tanto, esta sintaxis especial es suficiente para resolver los problemas de la redundancia y la ambigüedad.* *Nota:* Suponemos en este momento que cualquier intento de actualizar a V_DURANTE en forma tal que quede casi completamente fundida sobre DURANTE,

---

* Observamos que es necesario discutir el hecho de proporcionar una sintaxis de caso especial similar sólo para evitar el problema de redundancia y no el problema de la ambigüedad.

simplemente será rechazado. Sin embargo, vea la sección 22.10 para una explicación adicional de este punto

Por desgracia, las especificaciones KEY y COALESCED juntas todavía no son muy adecuadas, porque fallan para impedir que la varrel V_DURANTE contenga, por ejemplo, las dos tupias siguientes:

V2	Jones	10	París	*[d02,d08]*
V2	Jones	20	París	*[d07,dW]*

Aquí aparece el proveedor V2 como si tuviera un status de 10 y 20 en los días 7 y 8, un estado claramente imposible. En otras palabras, tenemos una *contradicción.*

Debe quedar claro que para impedir tal contradicción, debemos hacer cumplir una restricción de varrel (llamémosla *restricción C2)* que diga lo siguiente:

> Si dos tupias V_DURANTE distintas con el mismo valor V# tienen valores DURANTE *il* e ¿2, e *il* OVERLAPS *i2* es *verdadero,* entonces esas dos tupias deben ser idénticas, con la posible excepción de sus valores DURANTE.

Observe cuidadosamente que la restricción C2 *no* se hace cumplir manteniendo fundida a V_DURANTE sobre DURANTE (y obviamente no se hace cumplir por el hecho de que {V#, DURANTE} es una clave candidata). Pero supongamos que la varrel V_DURANTE se mantuviera **desdoblada** todo el tiempo sobre el atributo DURANTE. Entonces:

- La única clave candidata para esa forma desdoblada V_DURANTE UNFOLD DURANTE sería de nuevo la combinación de atributos {V#, DURANTE ((ya que en un momento dado, cualquier proveedor que esté actualmente bajo contrato, tiene sólo un nombre, un status y una ciudad).
- Por lo tanto, no puede haber dos tupias distintas que tengan el mismo valor V# y valores DURANTE "traslapados" (ya que todos los valores DURANTE son intervalos unitarios en V_DURANTE UNFOLD DURANTE, y dos tupias con el mismo valor V# y valores DU RANTE "traslapados" serían entonces duplicadas entre sí; de hecho, serían la misma tupia).

De aquí, se desprende que si hacemos cumplir la restricción de que {V#, DURANTE) es una clave candidata para V_DURANTE UNFOLD DURANTE, hacemos cumplir "automáticamente" la restricción C2. Por lo tanto, definamos una nueva cláusula I_KEY ("I_" por intervalo) que puede aparecer opcionalmente en lugar de la cláusula KEY usual en una definición de varrel como ésta:

```
VAR V_DURANTE BASE RELATION
  { V# V#, PROVEEDOR NOMBRE, STATUS INTEGER, CIUDAD CHAR,
    DURANTE INTERVAL ( DATE ) }
  I_KEY { V#, DURANTE UNFOLDED }
  COALESCED DURANTE ;
```

(lo que significa precisamente que {V#, DURANTE} es una clave candidata para V_DURANTE UNFOLD DURANTE).* Por lo tanto, esta sintaxis especial es suficiente para resolver el problema de la contradicción.

---

* Algunos escritores definen la semántica de LKEY de tal forma que se encargan también del problema de la redundancia. Encontramos este enfoque un poco ilógico y no lo adoptamos (en cualquier caso, es innecesario, debido a que COALESCED es claramente suficiente para manejar el problema de la redundancia).

Observe cuidadosamente que si {V#, DURANTE} es una clave candidata para V_DURANTE UNFOLD DURANTE, en realidad también lo es para V_DURANTE; éste es el hecho que nos permite eliminar la especificación KEY original para V_DURANTE a favor de la especificación I_KEY. Observe además que {V#, DURANTE} puede ser considerada como una **clave candidata temporal** en el sentido que explica la sección 22.3. Además, como acabamos de ver, esta clave candidata temporal en realidad es una clave candidata verdadera para la varrel que la contiene (a diferencia de las "claves candidatas temporales" que tratamos en la sección 22.3).

Por supuesto, si la sintaxis "I_KEY" es soportada por las claves candidatas, podemos esperar que también sea soportada por las claves externas. Por lo tanto la definición de VP_DURANTE, por ejemplo, podría incluir lo siguiente:

```
FOREIGN I_KEY { V#, DURANTE UNFOLDED } REFERENCES V_DURANTE ...
```

Lo que aquí pretendemos es que si VP_DURANTE muestra que el proveedor *Wx* pudo proporcionar alguna parte durante el intervalo *i,* entonces V_DURANTE debe mostrar que *Vx* estuvo bajo contrato a lo largo del intervalo *i.* Si esta restricción es satisfecha, entonces la combinación de atributos {V#, DURANTE} en la varrel VP_DURANTE puede ser considerada como una **clave externa temporal** en el sentido de la sección 22.3. (Sin embargo, todavía no es una clave externa verdadera en el sentido clásico.)

Hay un punto más a tratar con relación a la varrel V_DURANTE. Suponga que mantenemos esa varrel fundida todo el tiempo sobre DURANTE. Suponga también que de vez en cuando ejecutamos un procedimiento que vuelve a calcular el status de los proveedores que están contratados actualmente. Por supuesto, el procedimiento procura registrar los valores de status anteriores en V_DURANTE. Ahora, en ocasiones volver a calcularlos da como resultado que no hay cambio en el status. En ese caso, si el procedimiento trata de insertar a ciegas un registro de status anterior en V_DURANTE, ¡violará la restricción COALESCED! Para evitar tal violación el procedimiento tendrá que hacer una prueba especial para ver si no hay un "cambio en el status" y realizar un UPDATE adecuado en lugar del INSERT que se hace cuando *sí* cambia el status (vea el ejercicio 22.3 al final del capítulo). Por supuesto, en forma alterna podríamos decidir a fin de cuentas no mantener fundido a V_DURANTE sobre DURANTE; una solución que probablemente no es adecuada en este caso específico pero que pudiera serlo en otros casos.

22.10 OPERADORES DE ACTUALIZACIÓN QUE INVOLUCRAN INTERVALOS

En esta sección consideramos algunos problemas que se presentan con el uso de los operadores de actualización usuales INSERT, UPDATE y DELETE sobre una varrel temporal. Consideremos nuevamente a V_DURANTE, y supongamos que la definición de esa varrel incluye las especificaciones IJKEY y COALESCED como sugerimos en la sección anterior. Supongamos también (como es usual) que el valor actual de V_DURANTE es el que muestra la figura 22.4. Consideremos ahora los siguientes escenarios:

- *INSERT.* Suponga que descubrimos que el proveedor V2 estaba además bajo contrato durante el periodo del día 5 al día 6 (pero todavía se llamaba Jones, tenía status 10 y se encontraba en París en ese tiempo). No podemos simplemente insertar una tupia para tal efecto, porque si lo hacemos así, el resultado violaría la restricción COALESCED (¡ dos veces!). De hecho,

lo que debemos hacer es eliminar una de las tupias existentes V2 y actualizar la otra para poner el valor DURANTE a *[dO2,dlO]*.

- *UPDATE.* Suponga que descubrimos que en el día 9 el status de V2 fue incrementado tem poralmente a 20. Es bastante difícil hacer el cambio requerido, aunque parezca un UPDATE simple. En esencia, tenemos que dividir la tupia *[dO7, dlO]* de V2 en tres, con valores DU RANTE de *[dO7, dO8], [dO9, dO9]* y *[dlO, dlO]*, respectivamente, sin cambiar los demás valores, y luego reemplazar el valor de STATUS en la tupia *[dO9, dO9]* por el valor 20.
- *DELETE.* Suponga que descubrimos que el contrato del proveedor V3 terminó en el día 6 pero fue recontratado en el día 9. De nuevo, la actualización requerida no es trivial y requiere que la única tupia para V3 se divida en dos, con valores DURANTE de *[dO3, dO5] y [dO9, dlO]*, respectivamente.

¡Observe ahora que las soluciones que acabamos de delinear para estos tres problemas son *específicas del valor actual* de la varrel V_DURANTE (así como de las actualizaciones particulares deseadas)! Por ejemplo, considere el problema de inserción. En general, una tupia considerada para inserción puede ser simplemente insertada "tal como es" o puede necesitar ser fundida con una tupia "precedente", una tupia "siguiente" o (como en nuestro ejemplo) con ambas. En forma similar, en general es posible que las actualizaciones y eliminaciones requieran o no la "división" de tupias existentes.

Queda claro que la vida sería tremendamente complicada para los usuarios si estuvieran limitados a las operaciones INSERT, UPDATE y DELETE convencionales; algunas extensiones son muy necesarias. Éstas son algunas posibilidades:

- *INSERT.* De hecho, el problema de INSERT puede ser resuelto extendiendo simplemente la semántica de la especificación COALESCED de manera adecuada sobre la definición de la varrel. Para ser específicos, podemos permitir que el INSERT sea realizado en la forma normal y luego solicitar que el sistema haga cualquier (re)fundición necesaria después del INSERT. En otras palabras, la especificación COALESCED ya no define simplemente una restricción, sino que también especifica determinadas *acciones compensatorias* implícitas (similar, en alguna forma, a las acciones referenciales sobre las especificaciones de clave externa).

Sin embargo, desafortunadamente, la extensión de la semántica de COALESCED en esta forma no es suficiente por sí misma para resolver los problemas de UPDATE y DELETE (explique por qué no).

- *UPDATE.* El problema de UPDATE puede ser manejado extendiendo el operador UPDATE como sugiere el siguiente ejemplo:*

```
UPDATE V^DURANTE
       WHERE V# ■ V# ('V2')
       DURANTE INTERVAL ( [d09,<J09] )
       STATUS := 20 ;
```

Aquí la tercera línea —cuya sintaxis es básicamente <*nombre de atributo*> <*expresión de intervalo*>— especifica el atributo de intervalo sobre el cual se aplica la especificación

* Nuestra sintaxis es similar, pero no idéntica, a la propuesta en la referencia [22.3].

COALESCED (DURANTE en el ejemplo) y el valor del intervalo relevante *([dO9, dO9],* en el ejemplo). El UPDATE general puede ser entendido de la siguiente forma:

a. Primero, identifique las tupias para el proveedor V2.

b. Luego, entre esas tupias identifique aquellas donde el valor de DURANTE incluye el in tervalo *[dO9, dO9]* (por supuesto, debe haber como máximo una de estas tupias).

c. Si ninguna tupia es identificada, no se hace la actualización; en caso contrario, el sistema divide la tupia como sea necesario y realiza la actualización requerida.

DELETE. El problema de DELETE puede ser manejado extendiendo la operación DELETE en forma similar. Nuestro ejemplo se convierte en:

```
DELETE V_DURANTE
       WHERE V# = V# ('V3')
       DURANTE INTERVAL ( [d06,d08] ) ;
```

22.11 CONSIDERACIONES DE DISEÑO DE BASES DE DATOS

Nuestras varrels de ejemplo V_DURANTE y VP_DURANTE nos han servido bien hasta ahora, ilustrando claramente la necesidad de tipos de intervalo y la conveniencia de definir operadores especiales para manejar datos de intervalo. Ahora, esas dos varrels fueron "diseñadas" originalmente añadiendo simplemente atributos de intervalo a sus contrapartes de instantánea. En esta sección nos preguntamos si este enfoque de diseño en realidad es bueno. Más específicamente, sugerimos una *descomposición adicional* de determinadas varrels temporales (donde por "descomposición adicional" queremos decir la descomposición más allá de lo que requeriría la normalización clásica). De hecho, sugerimos la descomposición *horizontal* y la *vertical* en circunstancias adecuadas.

Descomposición horizontal

Nuestro ejemplo actual da por hecho, de manera razonable, que la base de datos contiene información histórica que comprende hasta el momento actual; sin embargo, también da por hecho que el momento actual está registrado como una fecha específica (es decir, el día 10) y esa suposición no es razonable. En particular, dicho enfoque sugiere que siempre que el tiempo siga su curso, por así decirlo, la base de datos es actualizada de alguna forma concordante (en nuestro ejemplo, sugiere que cada una de las apariciones de *dlO* sea reemplazada de alguna forma por *dll* en la medianoche del día 10). Un ejemplo diferente que involucre intervalos de una granularidad más fina, podría requerir que tales actualizaciones sucedieran con tanta frecuencia como (digamos) ¡cada milisegundo!

Algunas autoridades han sugerido el uso de un marcador especial, lo llamaremos *ahora,* que esté permitido en cualquier lugar en que está permitido un valor de punto. Bajo esta propuesta, el intervalo *[dO4, dlO],* por ejemplo, que muestra la figura 22.4 como valor de DURANTE para el proveedor VI en V_DURANTE, se convertiría en *[dO4, ahora].* Por supuesto, el valor real de dicho intervalo depende de cuándo lo vea, por decirlo así; en el día 14 sería *[dO4, dl4*\.

Otras autoridades consideran la introducción de *ahora* como una desviación imprudente de los conceptos en los que están basados los sistemas relaciónales. Observe que *ahora* es en realidad

una *variable;* por lo tanto, la propuesta conduce a la noción extraña —y diríamos lógicamente indefendible— de *valores* que contienen *variables.* Éstos son algunos ejemplos de preguntas que surgen a partir de esa noción y que tal vez quiera usted considerar:

- ¿Qué sucede con el intervalo *[ahora, di4]* en la media noche del día 14?
- ¿Cuál es el valor de *END([dO4, ahora])* en el día 14?; ¿es *dl4* o es *ahoral*

Creemos que es difícil dar respuestas coherentes a preguntas de esta naturaleza. Por lo tanto, preferimos buscar un enfoque que se mantenga dentro de los conceptos comprendidos ampliamente.

Ahora bien, a veces usaremos un "atributo DURANTE" para registrar información que se refiere tanto al futuro como al pasado (o en lugar de éste). Por ejemplo, tal vez queramos registrar la fecha futura en la que el contrato de un proveedor terminará o será considerado para su renovación. Si éste es el caso, podríamos usar el diseño de V_DURANTE de la figura 22.4. Sin embargo, es claro que este enfoque no siempre será aceptado. En particular, no será aceptado si DURANTE va a tener la interpretación de tiempo de *transacción* (vea la sección 22.2), ¡ya que los tiempos de transacción no pueden hacer referencia al futuro!

El problema general es que hay una diferencia importante entre (a) información histórica y (b) información que se refiere al estado actual de las cosas. La diferencia es ésta: para la información histórica se conocen las fechas inicial y final; por el contrario, para la información actual se conoce la fecha inicial pero (por lo general) no se conoce la final. Esta diferencia sugiere firmemente que debe haber dos varrels diferentes: una para el estado actual de las cosas y otra para la historia (ya que después de todo, en realidad hay dos *predicados* diferentes). En el caso de los proveedores, la varrel "actual" es V_A_PARTIR_DE (como muestra la figura 22.2) y en cambio, la varrel "historia" es V_DURANTE (como muestra la figura 22.4); salvo que las tupias cuyos valores DURANTE tengan tiempos finales de *dlO* serán omitidas, ya que en su lugar, la información relevante será registrada en V_A_PARTIR_DE.

Este ejemplo ilustra entonces la descomposición horizontal sugerida: una varrel con un atributo "a partir de" con valor de punto para el estado actual de las cosas, y una varrel con un atributo "durante" con valor de intervalo para la historia. De paso observamos que podemos usar *procedimientos disparados* para poblar la varrel de historia; por ejemplo, la eliminación de una tupia de V_A_PARTIR_DE podría disparar "automáticamente" la inserción de una tupia en V_DURANTE.

Podemos usar el operador relacional UNION para combinar la historia y los datos actuales en una sola relación:

```
V_DURANTE UNION ( EXTEND VJ\_PARTIR_DE
                  ADD INTERVAL [ A_PARTIRJ)E, TODAY() ]
                  AS DURANTE ) { ALL BUT A_PARTIR_DE }
```

Una desventaja posible de la descomposición horizontal ocurre cuando DURANTE tiene una interpretación de *tiempo válido* en lugar de tiempo de transacción. En este caso, ¡la historia es actualizable! Los operadores de actualización que describimos en la sección 22.10 podrían ser útiles aquí, pero habrá algunas ocasiones en que una revisión deseada tiene que afectar ambas varrels. Por ejemplo, suponga que descubrimos que el cambio más reciente sobre el status de un proveedor tiene errores. Entonces, tal vez no sólo necesitemos eliminar una tupia de V_DURANTE sino también actualizar otra en V_A_PARTIR_DE. Como otro ejemplo, si el cambio más reciente en el status fue correcto pero se hizo en el día erróneo, entonces de nuevo la revisión necesaria involucrará la actualización de ambas varrels.

Si VP_DURANTE es descompuesta de forma similar en VP_A_PARTIR_DE y VP_ DURANTE, necesitamos dar un repaso a las restricciones de clave externa. En el caso de VP_DURANTE ya vimos (en la sección 22.9) que la definición de varrel puede incluir lo siguiente:

```
FOREIGN ILKEY { V#, DURANTE UNFOLDED } REFERENCES V_DURANTE ...
```

Como dijimos en la sección 22.9, con esta especificación pretendemos que si el proveedor *Vx* aparece como que puede proporcionar alguna parte durante el intervalo *i*, entonces VDURANTE debe mostrar que *Wx* estaba contratado a lo largo del intervalo /. Y continuamos diciendo que {V#, DURANTE} en la varrel VP_DURANTE, debe ser considerada ahora como una clave externa temporal.

Sin embargo, en el caso de VP_A_PARTIR_DE, la clave externa correspondiente sólo es "semitemporal"; por lo tanto, todavía nos enfrentamos con el problema de tener que manejar la restricción engorrosa que mostramos en la sección 22.3:

```
CONSTRAINT AUM_VP_TO_V_FK
    IS_EMPTY ( ( ( V_A_PARTIR_DE RENAME A_PARTIRJ)E AS VA ) JOIN
                 ( VP_A_PARTIRJ3E RENAME AJ>ARTIR_DE AS VPA ) )
               WHERE VPA < VA ) ;
```

Por lo tanto, la descomposición horizontal conduce ciertamente a determinados problemas —el problema de las restricciones engorrosas y el problema de la actualización "simultánea" (como si así fuera) de las varrels actuales e históricas. Hasta el momento de la publicación de este libro, no hemos visto ninguna propuesta específica de abreviaturas que ayuden a resolver cualquiera de estos problemas. Tal vez sea necesario investigar más. Observamos que estos problemas no se presentan si permitimos que la varrel "DURANTE" incluya información tanto del futuro como del pasado y el presente (ya que es posible eliminar las varrels "A_PART1R_DE"), pero este enfoque requiere la predicción de las fechas finales futuras. El problema en cuestión tampoco se presenta en el enfoque propuesto en la referencia [22.4].

Descomposición vertical

Aun antes de que los datos temporales fueran estudiados —y antes de que SQL fuera inventado— algunos escritores estuvieron a favor de descomponer las varrels tanto como fuera posible, en lugar de hacerlo sólo cuando lo requiriera la normalización clásica. Por desgracia, algunos de estos escritores dañaron su causa proponiendo que los diseños de bases de datos consistieran completamente en varrels *binarías.* Una crítica a esta idea fue que en ocasiones también son necesarias las varrels unarias. Otra fue que algunas varrels de grado 3 o más, en realidad no son descomponibles (por ejemplo, considere la varrel VPY de la base de datos de proveedores, partes y proyectos).

Por otro lado, nuestra varrel V usual (no temporal) en realidad puede ser descompuesta aún más. Si tomamos en cuenta la veracidad de las frases "el nombre de VI es Smith", "el status de VI es 20" y "VI se encuentra en Londres", podemos concluir con seguridad la verdad del enunciado implicado por la primera tupia que muestra la figura 22.1 para V. Por lo tanto, podemos descomponer V en tres varrels binarias, cada una con V# como clave primaria.

La idea de descomponer al máximo (como si así fuera) está motivada por un deseo de reducir las verrels a los términos más simples posibles. Ahora, el caso de tal descomposición tal vez no es muy fuerte para la varrel V; sin embargo, es mucho más fuerte en el caso de la varrel V_DURANTE. El nombre, el status y la ciudad de un proveedor varían de manera independiente a lo

largo del tiempo. Además, probablemente también varían con distinta frecuencia. Por ejemplo, sería posible que el nombre de un proveedor nunca cambie, mientras que la ubicación del proveedor cambie ocasionalmente y el status correspondiente cambie muy frecuentemente, y probablemente sería incómodo tener que repetir el nombre y la ubicación cada vez que cambia el status. Además, la historia del nombre, la historia del status y la historia de la ciudad de un proveedor son probablemente conceptos más interesantes y digeribles que el concepto de una historia de nombre-status-ciudad combinados. Por lo tanto, proponemos la descomposición de VJ3URANTE en tres varrels históricas que se ven como ésta (en boceto):

```
V_NOMBRE_DURANTE     { V#, PROVEEDOR, DURANTE }
V_STATUS_DURANTE     { V#, STATUS, DURANTE } {
V_CIUDAD_DURANTE     V#, CIUDAD, DURANTE }
```

Las especificaciones I_KEY {V#, DURANTE UNFOLDED} y COALESCED DURANTE serían aplicadas a cada una de estas tres varrels. *Nota:* Probablemente quisiera incluir también la siguiente varrel de proveedores "maestra":

```
V#_DURANTE  { V#,  DURANTE  }
```

Esta varrel indicaría cuándo estuvieron bajo contrato los proveedores. De nuevo se aplicarían I especificaciones IJÍEY {V#, DURANTE UNFOLDED} y COALESCED DURANTE. Ad la combinación {V#, DURANTE} serviría como clave externa temporal para V_NOMBRE_DURANTE, V_STATUS_DURANTE y V_CIUDAD_DURANTE (y también para VP_Dl y correspondería a la clave candidata temporal {V#, DURANTE} de la varrel V_DURANTE. Aquí también hay otro punto que debemos considerar. Con V_DURANTE, tal como I definimos originalmente, tenemos que usar una expresión no muy trivial para obtener la his ria del status:

```
V_DURANTE { V#, STATUS, DURANTE } COALESCE DURANTE
```

Al mismo tiempo, la expresión para dar la (mucho menos interesante) historia combinada, ¡c sólo una simple referencia a varrel! Por lo tanto, en cierto sentido la descomposición suge "nivela el campo de juego" para las consultas, o más bien facilita la expresión de las más in teresantes y dificulta la expresión de las menos interesantes.

La necesidad de descomponer V_A_PARTIR_DE no es tan imperiosa. Observe en] ticular (de nuevo) que aunque podemos usar procedimientos disparados para poblar las I varrels históricas —por ejemplo, la eliminación de una tupia de V_A_PARTIR_DE podría **disparar** "automáticamente" las actualizaciones a V#_DURANTE, V_NOMBRE_DUF V_STATUS_DURANTE y V_CIUDAD_DURANTE—, no hay necesidad de descomponer! V_A_PARTIR_DE para lograr tales efectos.

22.12 RESUMEN

Comenzamos este capítulo con una referencia a la creciente necesidad de que las bases de (contengan datos **históricos** así como actuales. Mostramos que la representación de datos histó usando solamente **marcas de tiempo** conduce a dificultades severas —en particular hace i determinadas restricciones y consultas sean muy difíciles de manejar— y tratamos el uso de **1 intervalos** escalares ("encapsulados") como un mejor enfoque. Para ser más específicos, exp

camos el uso del **generador de tipo** INTERVAL junto con varios **operadores** nuevos para manejar datos de intervalo (aunque le recordamos que casi todos estos operadores en realidad sólo son abreviaturas). Los intervalos y sus operadores relacionados sirven para más propósitos que los simples datos temporales en sí; a pesar del hecho de que nuestro ejemplo actual estuvo basado específicamente en el tipo INTERVAL(DATE). Mostramos ejemplos de **relaciones temporales** (y explicamos las **varrels temporales)** con atributos de este tipo en particular.

Un tipo de intervalo debe ser definido sobre un **tipo de punto** subyacente y es necesario especificar (de alguna forma) una **precisión** asociada para ese tipo de punto. También es necesario definir una **función sucesora** para este tipo de punto y esa precisión.

Los operadores que describimos incluyen operadores sobre intervalos en sí, operadores sobre conjuntos de intervalos y operadores sobre relaciones temporales. Los operadores sobre intervalos en sí incluyen START, END y los **operadores de Alien.** Los operadores sobre conjuntos de intervalos incluyen **UNFOLD** y **COALESCE** (vea el siguiente párrafo). Los operadores sobre relaciones temporales incluyen las **versiones relaciónales** de UNFOLD y COALESCE (de nuevo, vea el siguiente párrafo). También tratamos determinados operadores de **actualización** especializados y determinadas **restricciones** especializadas para las varrels temporales ("claves temporales"). Mostramos que la mayoría de estos nuevos operadores y restricciones pueden ser considerados, en efecto, como contrapartes temporales de construcciones familiares.

Vimos dos *formas canónicas* importantes para conjuntos de intervalos del mismo tipo, la forma **desdoblada** y la forma **fundida.** Un conjunto de intervalos de tipo INTERVAL(7P) está en forma desdoblada cuando cada uno de los intervalos del conjunto es un *intervalo unitario;* es decir, un intervalo que sólo contiene un *punto* (donde un punto es un valor del tipo de punto *TP* subyacente). Un conjunto de intervalos de tipo INTERV\h{TP)* está en forma fundida si no hay dos intervalos distintos que se *traslapen* o *reúnan* en el conjunto. Ambas formas canónicas tienen la ventaja de evitar determinados tipos de redundancia. La forma fundida maximiza lo conciso y tiene muchas ventajas psicológicas; en cambio, la forma desdoblada es la más fácil de operar (lo que evita la necesidad de las restricciones especiales y los operadores de actualización que tratamos en las secciones 22.9 y 22.10). Mostramos la forma en que el concepto de estas formas canónicas se extiende a relaciones con atributos de intervalo, lo que conduce a los nuevos operadores relaciónales importantes **UNFOLD** y **COALESCE.** Usamos esos operadores para definir equiivalente temporales de los operadores relaciónales familiares de proyección y diferencia.

Por último, dirigimos nuestra atención hacia determinados asuntos de **diseño de base de datos** que tienen que ver con la **descomposición horizontal** y **vertical** de determinadas varrels temporales.

EJERCICIOS

22.1

a. El tipo de datos VARCHAR(3) de SQL consiste en todas las cadenas de hasta tres caracteres de un *conjunto predeterminado de caracteres,* el cual suponemos que es ASCII. ¿Cree usted que INTERVAL(VARCHAR(3)) sería un tipo de intervalo aceptable?

b. Si su respuesta al punto a. es sí, exprese el intervalo cerrado-abierto ['p','q') de este tipo en notación cerrada-cerrada.

22.2 En la sección 22.8 definimos el operador de *diferencia temporal* I_MINUS. Los operadores de unión temporal (IJJNION) e intersección temporal (I_INTERSECT) pueden ser definidos en forma similar. Dé las definiciones adecuadas.

22.3 Suponga que la varrel V_DURANTE está restringida para estar fundida en DURANTE, y suponga que necesita ser actualizada para que refleje el hecho de que el proveedor V1 tenía un status de 20 desde el día 11 hasta el 15. Dé una instrucción que tenga el efecto deseado. No dé por hecho que los operadores de actualización están extendidos como sugiere la sección 22.10. Sin embargo, suponga que V_DURANTE contiene información acerca del proveedor VI hasta el día 10 y no des pués del día 10. No haga ninguna suposición sobre cuál podría ser el status grabado de VI en el día 10.

22.4 En este capítulo hemos mostrado cómo ciertos operadores que se aplican a los intervalos en gene ral pueden ser especialmente útiles para los intervalos de tiempo. Sugiera algunas otras aplicaciones posibles de estos operadores, donde se involucren intervalos que no sean de tiempo.

22.5 Sugiera algunos ejemplos realistas de relaciones que tengan más de un atributo de intervalo, tem poral o de cualquier otro tipo.

22.6 Considere nuevamente la varrel V_DURANTE. En un momento dado, si hay algún proveedor en ese momento, existe algún status *smax* tal que ningún proveedor tenga un status que en ese mismo momento sea superior a *smax*. Use los operadores que explicamos en este capítulo para obtener la relación fundida en la cual cada valor de status que alguna vez fue *smax*, esté emparejado con los in tervalos durante los cuales fue el valor *smax*.

22.7 AP es una relación que tiene los atributos, NOMBRE, ALTURA y PESO, dada la altura y el peso de determinadas personas. Escriba una consulta que muestre, para cada peso registrado, todos los ran gos de altura tales que para cada altura en ese rango haya al menos una persona de esa altura y de ese peso.

22.8 Considere una relación *R* con dos atributos de intervalo distintos *II* e *12*. Pruebe o desapruebe las siguientes aseveraciones:

a. `(fl UNFOLD if) UNFOLD` *12 m* `(fl UNFOLD` *12)* `UNFOLD` *11*

b. `(fl COALESCE I")) COALESCE` *12 m* `(fl COALESCE` *12)* `COALESCE` *11*

22.9 ¿Puede pensar un ejemplo de una varrel con un atributo de intervalo que *no* se quiera tener en forma fundida?

22.10 Investigue la posibilidad de extender el concepto de "clave externa temporal" para incluir acciones referenciales tales como la eliminación en cascada.

REFERENCIAS Y BIBLIOGRAFÍA

En lugar de dar lo que por mucho sería una lista muy larga de referencias, simplemente ponemos a su consideración la bibliografía completa que está en la referencia [22.2], véala.

22.1 J. F. Allen: "Maintaining Knowledge about Temporal Intervals", *CACM16*, No. 11 (noviembre, 1983).

22.2 Opher Etzion, Sushil Jajodia y Suryanaryan Sripada (eds.): *Temporal Databases: Research and Practice.* Nueva York, N.Y.: Springer Verlag (1998).

Es una antología que presenta lo más avanzado en 1997, así como una referencia primaria excelente para estudios adicionales. La parte 4 es *una referencia general* e incluye una bibliografía completa, así como la versión de febrero de 1998 del *glosario consensado de conceptos de bases de datos temporales.* La parte 2 trata sobre *lenguajes de consultas temporales* e incluye un artículo titulado "Propuestas de tiempo válido y tiempo de transacción: aspectos sobre el diseño del lenguaje", donde el autor original del presente capítulo (Hugh Darwen) argumenta en contra del enfoque tomado en TSQL2 y afirma haber encontrado fallas importantes en la especificación TSQL2 [22.4]. También incluye un artículo de David Toman titulado "Extensiones temporales basadas en punto para SQL y su implementación eficiente", el cual propone una extensión a SQL basada en puntos en vez de intervalos. Esta idea hace que surjan algunas preguntas interesantes relativas

a la implementación. Las respuestas a esas preguntas también pueden ser importantes para los lenguajes basados en intervalos, ya que los *intervalos unitarios* resultantes de UNFOLD son "prácticamente" puntos (además *son* puntos en IXSQL; vea el comentario a la referencia [22.3]).

22.3 Nikos A. Lorentzos y Yannis G. Mitsopoulos: "SQL Extension for Interval Data", *IEEE Transactions on Knowledge and Data Engineering 9,* No. 3 (mayo-junio, 1997).

Muchas de las ideas que tratamos en el presente capítulo están basadas en el trabajo reportado en este artículo. Al igual que la referencia [22.2], el escrito también incluye muchas referencias adicionales útiles.

Antes de presentar su extensión propuesta para SQL, los autores definen un *álgebra relaciona! extendida con intervalos.* La extensión propuesta para SQL se llama IXSQL (que a veces se pronuncia "nueve SQL") y no es específica para intervalos de tiempo. Debido a que las palabras reservadas INTERVAL y COALESCE ya se usan en SQL para propósitos diferentes a los que se están tratando, los autores proponen PERIOD (incluso para los intervalos que no son temporales) y NORMALISE (observe la ortografía) en su lugar. Como dijimos en el comentario a la referencia [22.2], el UNFOLD de IXSQL difiere del nuestro en que produce puntos en vez de intervalos unitarios. Por consecuencia, Lorentzos y Mitsopoulos proponen un operador inverso, FOLD, que convierte puntos en intervalos unitarios y luego los funde. UNFOLD, FOLD y NORMALISE son propuestos en la forma de cláusulas adicionales para la construcción ya familiar SELECT-FROM-WHERE. Es interesante observar que la cláusula propuesta NORMALISE ON no sólo es escrita al último sino que —apartándose del SQL (vea el apéndice B)— también es ejecutada al final; es decir, la salida de la cláusula SELECT es la entrada para la cláusula NORMALISE ON (por buenas razones).

22.4 Richard T. Snodgrass (ed.): *The Temporal Query Language TSQL2.* Dordrecht, Países Bajos: Kluwer Academic Pub. (1995).

TSQL2 es un conjunto de extensiones temporales propuestas para SQL. En gran parte, el comité TSQL2 rechaza el enfoque general de operadores escalares y relaciónales sobre intervalos a favor de algo que es más adecuado en determinados casos especiales. Por lo tanto, en vez de soportar simplemente un generador de tipo de intervalo y operadores asociados, propone diversos tipos especiales de tablas: *tablas de instantánea, tablas de estado de tiempo válido, tablas de evento de tiempo válido, tablas de tiempo de transacción, tablas de estado bitemporales y tablas de evento bitemporales.*

- Una *tabla de instantánea* es una tabla SQL antigua, que incluye posiblemente columnas de tipo de datos PERIOD (como en IXSQL [22.3], esta palabra reservada es usada en vez de INTER VAL, debido a que SQL ya usa INTERVAL para otro propósito).
- Se dice que los demás tipos de tabla tienen *soporte temporal;* donde soporte temporal implica la existencia de uno o dos *elementos temporales* para cada fila. Un elemento temporal es un con junto de *marcas de tiempo,* donde una marca de tiempo es un valor PERIOD o algún otro tipo de dato fecha-hora. (Observe por lo tanto, que el término "marca de tiempo" no se está usando aquí en el sentido convencional del SQL/92.)

Los elementos temporales que consisten en valores PERIOD están especificados para que sean fundidos.* Los elementos temporales no aparecen como columnas normales, sino que en vez de ello, son accedidos por medio de operadores de propósito especial.

---

* La versión de TSQL2 que en 1996 fue propuesta ante el ISO (aunque no fue aceptada) para su inclusión en el estándar SQL, difiere con respecto a la versión descrita en la referencia [22.4], en que las tablas con soporte temporal siempre eran"no anidadas" (lo que quiere decir que cada elemento temporal era una marca de tiempo única y no un conjunto de marcas de tiempo). Tampoco especificaba cuándo se realizaba la fundición.

Éste es un estudio rápido de los diversos tipos de tabla "con soporte temporal":

- En las *tablas de estado de tiempo válido* y las *tablas de tiempo de transacción* cada marca de tiempo es un valor PERIOD.
- En las *tablas de evento de tiempo válido* cada marca de tiempo válido es un valor de algún tipo de dato fecha-hora.
- Una *tabla bitemporal* es aquella que (a) es una tabla de tiempo de transacción y (b) una tabla de estado de tiempo válido o una tabla de evento de tiempo válido. Cada renglón de una tabla bitem poral tiene dos elementos bitemporales, uno para el tiempo de transacción y otro para el tiempo válido. Por lo tanto, una tabla bitemporal puede ser operada como una tabla de tiempo de transac ción o una tabla de tiempo válido.

TSQL2 está motivado firmemente por una noción que llama *compatibilidad temporal hacia arriba*. La idea es tener la posibilidad de añadir "soporte temporal" a una tabla base existente, convirtiendo por lo tanto a esa tabla base de una tabla de instantánea en una tabla temporal de algún tipo. A partir de ahí, todas las operaciones SQL normales sobre la tabla base son interpretadas como operaciones sobre la versión de instantánea actual de esa tabla,* aunque ahora pueden tener nuevos efectos laterales. En particular, las actualizaciones y eliminaciones sobre la versión de instantánea actual dan como resultado la retención de las versiones antiguas de esos renglones como renglones con elementos temporales.

La gran ventaja del enfoque del TSQL2 aumenta junto con lo que llaman operaciones *en secuencia*. Una operación en secuencia es aquella que está expresada como una operación sobre una instantánea de la base de datos —por lo general la instantánea actual— pero que es ejecutada, como si lo fuera, en *cada una* de las instantáneas. Por ejemplo, el resultado de una consulta en secuencia sobre tablas de tiempo válido, es una tabla de tiempo válido. La propia consulta es expresada como si fuera una consulta contra una base de datos de instantánea actual, con la incorporación de una sola palabra reservada para indicar que se trata de una consulta en secuencia. El programa de aplicación que hace una consulta de éstas tiene que hace provisiones especiales para acceder a las marcas de tiempo de los renglones resultantes.

Las operaciones que no pueden ser expresadas como operaciones en secuencia requieren del uso ocasional de una sintaxis bastante rara. Como una consecuencia de la falla de SQL para soportar tablas que no tienen columnas, TSQL2 tiene la restricción de que una tabla con soporte temporal debe tener al menos una columna normal además de sus elementos temporales. Por lo tanto, las consultas como la que se hace para mostrar los periodos durante los cuales estaba contratado al menos un proveedor de París, no pueden ser expresadas en forma de secuencia.

RESPUESTAS A EJERCICIOS SELECCIONADOS

22.1 a. Probablemente no (aunque *pudiera* serlo), b. La cadena de caracteres que precede inmediatamente a 'q' depende de qué carácter específico está al final en la secuencia de ordenamiento utilizada (vea la referencia [4.19] para una explicación de las secuencias de ordenamiento SQL). Si ese último carácter es 'Z', entonces la respuesta es ['p', 'pZZ']. *Nota:* Le recordamos que, en términos estrictos, el "(3)" que está en VARCHAR(3) es mejor visto no como parte del tipo, sino como una *restricción de integridad*.

22.2 Primero, *Rl* IJJNION *R2* ON *A* es equivalente a

```
( R1 UNION R2 ) COALESCE A
```

---

* De hecho, hay otra diferencia entre TSQL2 (tal como está definido en la referencia [22.4]) y la versión propuesta ante ISO. La referencia [22.4] requiere de la palabra reservada SNAPSHOT después de SELECT para indicar que una consulta es en contra del estado actual de cada una de las tablas a las que hace referencia; la versión propuesta ante ISO no lo requiere.

Observe que no hay necesidad de desdoblar *Rl* y *R2* sobre *A* antes de formar la unión (¿por qué no?). Segundo, *Rl* IJNTERSECT *R2* ON *A* es equivalente a

```
( ( EXTEND ( ( fl) RENAME A AS A1 ) JOIN
             ( R2 RENAME A AS A2 ) WHERE
             Al OVERLAPS A2 )
    ADD ( A1 INTERSECT A2 ) AS A ) { ALL BUT A1, A2 } )
COALESCE A
```

De nuevo, no hay necesidad de desdoblar *RJ* y *R2* sobre *A.* Además, si *Rl y R2* están de hecho fundidas sobre A, tampoco hay necesidad del paso de fusión final (¿por qué no?).

Las versiones temporales de los demás operadores relaciónales (por ejemplo, I_JOIN) pueden ser definidas en forma similar. En particular, las versiones especiales de UPDATE y DELETE que describimos en la sección 22.10 hacen uso tácito de una versión temporal de restricción. *Nota:* Tal vez debemos explicar por qué enfatizamos la abreviatura de I_MINUS específicamente en el cuerpo del capítulo. El asunto es que I_MINUS involucra el desdoblado, mientras que (como acabamos de ver) los demás operadores relaciónales temporales generalmente no lo involucran,* y es preferible tener la posibilidad de solicitar o realizar tales desdoblamientos siempre que sea posible.

22.3 Ésta es una solución posible:

```
IF IS_EMPTY ( V_DURANTE WHERE V# ■ V# ('V1') AND
STATUS = 20 AND END ( DURANTE ) = dW ) THEN INSERT
INTO V_DURANTE
          ( EXTEND ( V_DURANTE WHERE V# = V# ('V1')
                                AND END ( DURANTE ) = dW )
                                    { ALL BUT DURANTE }
            ADD INTERVAL ( [ d11, d15 ] ) AS DURANTE ) ;
ELSE UPDATE VJXJRANTE WHERE V# = V# ('V1')
                      AND END ( DURANTE ) ■ d10 DURANTE :=
            INTERVAL ( [ START ( DURANTE ), d15 ] ) ;
```

22.4 Los animales varían de acuerdo con el rango *de frecuencias* de ondas de luz y sonido a las que sus ojos y oídos son receptivos. Hay diversos fenómenos naturales que pueden ser medidos en ran gos de *profundidad* del suelo o del mar, o de *altura* sobre el nivel del mar. El hecho de que el té se tome entre las 4:00 y las 5:00 horas de la tarde es una observación temporal, pero es una significati vamente diferente en tipo con respecto a los ejemplos que damos en el cuerpo del capítulo (¿en qué manera exactamente?). No hay duda que usted ha podido pensar en muchos ejemplos similares sobre los cuales es posible basar aplicaciones de bases de datos interesantes.

22.5 ¡Los animales varían de acuerdo con el rango de frecuencias de ondas de luz y sonido a las que sus ojos y oídos son receptivos! Además, tan pronto como juntamos dos relaciones temporales *Rl [A, B]* y *R2{A, C},* en donde *By C* son atributos de intervalo, obtenemos un resultado que (aunque sea uno intermedio) tiene más de un atributo de intervalo.

```
22.6  WITH VPJ5URANTE UNFOLD DURANTE AS VP.DESDOBLADA :
      ( SUMMARIZE VPJ)ESD0BLADA PER VP_DESD0BLADA { DURANTE }
                 ADD MAX ( STATUS ) AS SMAX ) COALESCE DURANTE

22.7 ( ( EXTEND AP { ALTURA, PESO }
        ADD INTERVAL ( [ ALTURA, ALTURA ] ) AS RA )
                                      { PESO, RA } ) COALESCE RA
```

---

* SUMMARIZE es una excepción. Vea la respuesta al ejercicio 22.6.

22.8 Fácilmente se ve que la aseveración a. es válida. Pero la aseveración b. no lo es, como ahora veremos. Sea R de la siguiente manera:

11	12
[d01,d01] *[d01,d02]* *[d03,d04]* *[d04,d04]*	*[d08,d09]* *[d08,d09]* *[d08,d08]* *[d08,d08]*

Entonces, el resultado U de (R UNFOLD I I) UNFOLD 12 y (R UNFOLD 12) UNFOLD II es:

11	12
[d01, d01]	*[d08, d08]*
[dei, d01] *[d02, d02]* *[d02, d02)* *[d03, d03]* *[d04, d04]*	*[d09, d09]* *[d08, d08]* *[d09, d09]* *[d08, d08]* *\d08, d08]*

Sin embargo, el resultado de (U COALESCE I I) COALESCE 12 es:

11	12
[d01,d02] *[d03,d04]*	*[d09,d09]* *[d08,d08]*

Mientras que el resultado de (U COALESCE 12) COALESCE II es:

11	12
[d0i,d02] *[d03,d04*	*[d08, d09]* *[d08,d08]*

Le sugerimos que revise estos resultados, escribiendo los resultados de U COALESCE II y U COALESCE 12, y luego fundiendo estos resultados intermedios sobre 12 e II, respectivamente. También le sugerimos las siguientes abreviaturas:

- *R* UNFOLD *11, 12* = (fl UNFOLD *11)* UNFOLD *12*
- *R* COALESCE *11, 12 s* (fl COALESCE *11)* COALESCE *12*

CAPITULO 23

Bases de datos basadas en la lógica

23.1 INTRODUCCIÓN

A mediados de los años ochenta comenzó a emerger en la comunidad de investigación sobre bases de datos, una tendencia importante hacia los **sistemas de bases de datos basados en la lógica.** En la literatura de investigación, comenzaron a aparecer expresiones tales como *base de datos lógica, DBMS inferencial, DBMS experto, DBMS deductivo, base de conocimiento, KBMS {sistema de administración de base de conocimiento), lógica como un modelo de datos, procesamiento de consultas recursivas,* etcétera. Sin embargo, no siempre es fácil relacionar tales términos, ni las ideas que representan, con los términos y conceptos familiares de base de datos; tampoco es fácil comprender la motivación que hay detrás de una investigación desde una perspectiva de base de datos tradicional. Existe una necesidad clara de explicar toda esta actividad en términos de las ideas y principios de las bases de datos convencionales. Este capítulo intenta satisfacer esta necesidad.

Por lo tanto, nuestro propósito es explicar de qué tratan los sistemas basados en la lógica desde el punto de vista de alguien que está familiarizado con la tecnología de bases de datos tradicionales, aunque tal vez no tanto con la lógica como tal. Por lo tanto, conforme presentemos cada idea de la lógica la explicaremos en términos de bases de datos convencionales, siempre que sea posible o adecuado. (Por supuesto, ya hemos explicado ciertas ideas de la lógica, en especial en nuestra descripción del cálculo relacional en el capítulo 7. El cálculo relacional está basado directamente en la lógica. Sin embargo, como veremos, hay más en los sistemas basados en la lógica que simplemente el cálculo relacional.)

La estructura del capítulo es la siguiente. Después de esta sección introductoria, la sección 23.2 presenta un breve panorama del tema enriquecido con un poco de historia. A continuación, las secciones 23.3 y 23.4 proporcionan un tratamiento elemental (y muy simplificado) del *cálculo proposicional* y del *cálculo de predicados,* respectivamente. Luego, la sección 23.5 presenta la llamada vista de una base de datos por la *teoría de las demostraciones,* y la sección 23.6 desarrolla las ideas de esa sección para explicar lo que significa el término *DBMS deductivo.* Después, la sección 23.7 trata algunos enfoques al problema del *procesamiento de consultas recursivas.* Por último, la sección 23.8 proporciona un resumen y algunos comentarios finales.

23.2 PANORAMA GENERAL

La investigación sobre la relación entre la teoría de base de datos y la lógica, se remonta por lo menos a finales de los años setenta (si no es que antes); vea por ejemplo, las referencias [23.5], [23.7] y [23.13]. Sin embargo, el principal estímulo para que el interés sobre el tema se expandiera recientemente de manera considerable, parece haber sido la publicación —en 1984— de un artículo muy destacado de Reiter [23.15]. En ese artículo, Reiter caracterizó la percepción

tradicional de los sistemas de bases de datos como una **teoría de modelos,** lo que significa en términos generales que:

a. La base de datos puede ser vista, en cualquier momento dado, como un conjunto de relacio nes explícitas (es decir, base) donde cada una contiene un conjunto de tupias explícitas, y
b. La ejecución de una consulta puede ser considerada como la evaluación de alguna fórmula especificada (es decir, una expresión con valor de verdad) sobre esas relaciones y tupias explícitas.

Nota: En la sección 23.5, definiremos el término "teoría de modelos" de forma más precisa.

Luego Reiter continúa explicando que es posible tener una vista alterna por **la teoría** de demostraciones y además preferible en ciertos aspectos. En esa vista alterna (de nuevo, en general):

a. La base de datos es vista en cualquier momento dado como un conjunto de **axiomas** (axio mas "base", que corresponden a valores en dominios y tupias en relaciones base, y además determinados axiomas "deductivos", que explicaremos), y
b. La ejecución de una consulta es considerada como la demostración de que alguna fórmula especificada es una consecuencia lógica de esos axiomas; en otras palabras, demostrar que es un **teorema.**

Nota: En la sección 23.5, definiremos de manera más precisa el término "teoría de demostraciones", aunque en este momento puede servir señalar que la vista por la teoría de las demostraciones está muy cercana a la caracterización de una base de datos que dimos en el capítulo 1, sección 1.3 (subsección "Datos y modelos de datos") como *una colección de proposiciones verdaderas* [1.2]. Vale la pena dar un ejemplo. Considere la siguiente consulta del cálculo relacional contra la base de datos usual de proveedores y partes:

```
VPX WHERE VPX.CANT > 250
```

(por supuesto, en este caso VPX es una variable de alcance sobre los envíos). En la interpretación tradicional —es decir, por la teoría de modelos— examinamos las tupias de envíos una por una, evaluando la fórmula "CANT > 250" para cada una; el resultado de la consulta consiste entonces en esas tupias de envíos para las cuales la fórmula da como resultado *verdadero*. En la interpretación por la teoría de demostraciones, por el contrario, consideramos a las tupias de envíos (además de otros conceptos) como *axiomas* de una determinada **"teoría lógica";** luego aplicamos técnicas de demostración de teoremas con el fin de determinar para cuáles de los valores posibles de la variable de alcance VPX, la fórmula "VPX.CANT > 250" es una consecuencia lógica de esos axiomas dentro de la misma teoría. El resultado de la consulta consiste entonces simplemente en esos valores particulares de VPX.

Por supuesto, este ejemplo es extremadamente simple, de hecho es tan simple que es posible que le sea difícil ver cuál es realmente la diferencia entre las dos interpretaciones. Sin embargo, el punto es que el mecanismo de razonamiento empleado en esta demostración (en la interpretación por la teoría de demostraciones) puede por supuesto ser mucho más sofisticado que nuestro ejemplo simple; además, puede manejar determinados problemas que están más allá de las posibilidades de los sistemas relaciónales clásicos, como veremos. Además, la interpretación por la teoría de demostraciones lleva consigo un conjunto atractivo de características adicionales [23.15]:

- **Uniformidad de representación.** Nos permite definir un lenguaje de base de datos en el que los valores en dominios, tupias en relaciones base, "axiomas deductivos", consultas y restricciones de integridad están todos representados esencialmente en la misma forma uniforme.

Uniformidad operacional. Proporciona una base para un ataque unificado sobre una variedad de problemas aparentemente distintos, incluyendo optimización de consultas (en especial la optimización semántica), cumplimiento de restricciones de integridad, diseño de base de datos (teoría de dependencias), pruebas de la corrección de programas y otros problemas.

Modelado semántico. Proporciona las bases sólidas sobre las cuales construir una variedad de extensiones "semánticas" para el modelo básico.

Aplicación extendida. Por último, proporciona una base para manejar determinados asuntos en los que tradicionalmente han tenido dificultades los enfoques clásicos como la *información disyuntiva* (por ejemplo, "El proveedor V5 proporciona la parte Pl o P2, pero no se sabe cuál").

Axiomas deductivos

Proporcionamos una explicación breve y preliminar del concepto, que ya hemos mencionado algunas veces, de un **axioma deductivo** (también conocido como **regla de inferencia).** En esencia, un axioma deductivo es una regla por la cual, dados ciertos hechos, podemos deducir hechos adicionales. Por ejemplo, dados los hechos "Ana es la madre de Bety" y "Bety es la madre de Celia", hay un axioma deductivo obvio que nos permite deducir que Ana es la abuela de Celia. Por lo tanto, si nos adelantamos un momento, podríamos imaginar un *DBMS deductivo* en el cual los dos hechos dados estén representados como tupias en una relación, entonces:

MADRE DE

MADRE	HIJA
Ana	Bety
Bety	Celia

Estos dos hechos representan **axiomas** base para el sistema. Supongamos también que el axioma deductivo le ha sido especificado formalmente al sistema de alguna manera, por ejemplo, de la siguiente:

```
IF  MADRE_DE  (  X,  y  )
AND  MADRE_DE  (  y,  z  )
THEN ABUELA_DE ( x, z ) ;
```

(sintaxis hipotética y simplificada). Ahora el sistema puede aplicar la regla expresada en el axioma deductivo sobre los datos que están representados por los axiomas base, en una forma que explicaremos en la sección 23.4, para deducir el resultado de ABUELA_DE (Ana, Celia). Por lo tanto, los usuarios pueden hacer consultas como "¿quién es la abuela de Celia"? o "¿quiénes son las nietas de Ana?" (o más específicamente, "¿de quién es abuela Ana?").

Tratemos ahora de relacionar las ideas anteriores con los conceptos de bases de datos tradicionales. En términos tradicionales, el axioma deductivo puede ser visto como una *definición de vista,* por ejemplo:

```
VAR ABUELA_DE VIEW
  ( MX.MADRE AS ABUELA, MY.HIJA AS NIETA )
    WHERE MX.HIJA = MY.MADRE ;
```

(aquí usamos deliberadamente un estilo de cálculo relacional; MX y MY son variables de alcance sobre MADRE_DE). Consultas como las que mencionamos anteriormente pueden ahora ser enmarcadas en términos de esta vista:

```
AX.ABUELA WHERE AX.NIETA - NOMBRE ( 'Celia' )

AX.NIETA WHERE AX.ABUELA ■ NOMBRE ( 'Ana' )
```

(AX es una variable de alcance sobre ABUELA_DE).

Por lo tanto, todo lo que en realidad hemos hecho hasta el momento es presentar una sintaxis diferente y una interpretación diferente sobre cosas que ya nos son familiares. Sin embargo, en secciones posteriores veremos que existen de hecho algunas diferencias importantes (que no ilustran estos ejemplos simples) entre los sistemas basados en la lógica y los DBMSs más tradicionales.

23.3 CALCULO PROPOSICIONAL

En esta sección y la siguiente presentamos una introducción muy breve a algunas de las ideas básicas de la lógica. La presente sección considera al *cálculo proposicional* y la siguiente considera al *cálculo de predicados.* Sin embargo, hacemos notar inmediatamente que, en lo que a nosotros se refiere, el cálculo proposicional no es tan importante como para ser un fin por sí mismo; el propósito principal de la presente sección en realidad es allanar simplemente el camino para comprender la siguiente. El propósito de las dos secciones juntas es proporcionar una base sobre la cual construir en el resto del capítulo.

Se supone que usted ya está familiarizado con los conceptos básicos del álgebra de Boole. Para efectos de referencia, planteamos aquí determinadas leyes del álgebra de Boole que necesitaremos después:

Leyes distributivas:

```
f AND ( g OR  h ) s f  f AND g ) OR  ( f AND h
OR  ( g AND h ) s     f OR  g ) AND ( f OR  ft
```

Leyes de De Morgan:

```
NOT     f AND g ) ■ NOT f OR NOT g
NOT     f OR g ) s NOT f AND NOT g
```

Aquí/, g y *h* son cualquier expresión lógica (con valores de verdad).

Ahora pasemos a la lógica misma. La lógica puede ser definida como un **método formal de razonamiento.** Puesto que es formal, puede ser usada para realizar tareas formales como demostrar la validez de un argumento mediante el simple análisis de la estructura de ese argumento como una secuencia de pasos (es decir, sin poner atención alguna al significado de esos pasos). En particular, debido a que es formal, puede por supuesto ser *mecanizada;* es decir, puede ser programada y por lo tanto aplicada por la máquina.

El cálculo proposicional y el cálculo de predicados son dos casos especiales de la lógica en general (de hecho, el primero es un subconjunto del segundo). El término "cálculo", a su vez.es simplemente un término general que se refiere a cualquier sistema de cálculo simbólico; en el caso particular que tenemos en la mano, el tipo del cálculo involucrado es el del valor de verdad —*verdadero o falso*— de determinadas fórmulas o expresiones.

Términos

Comenzamos suponiendo que tenemos alguna colección de objetos, llamados **constantes,** sobre los cuales podemos hacer declaraciones de diversos tipos. En la jerga de las bases de datos, las *constantes* son los valores en los dominios subyacentes y una *declaración* puede ser, por ejemplo, una expresión condicional como "3 > 2". Definimos un **término** como una declaración que involucra dichas constantes y:

a. No involucra ningún conector lógico (vea más adelante) o está encerrada entre paréntesis, y
b. Da como resultado inequívocamente *verdadero o falso.*

Por ejemplo, "el proveedor VI se encuentra en Londres", "el proveedor V2 se encuentra en Londres" y "el proveedor VI proporciona la parte Pl" son todos términos (y dan como resultado *verdadero, falso* y *verdadero,* respectivamente, tomando en cuenta nuestros valores de datos de muestra usuales). Por el contrario, "el proveedor VI proporciona la parte *p"* (donde *p* es una variable) y "el proveedor V5 proporcionará la parte Pl en algún tiempo futuro" no son términos, debido a que no dan como resultado *verdadero o falso* inequívocamente.

Fórmulas

A continuación, definimos el concepto de una **fórmula.** Las fórmulas del cálculo proposicional —y más generalmente, del cálculo *de, predicados*— son usadas en los sistemas de base de datos en la formulación de consultas (entre muchas otras cosas).

```
<fórmula>  | <término> NOT <términc»
           | <término> AND <fórmula>
           | <término> OR  <fórmula>
           | <término> =► <fórmula>

    ::=    <fórmula atómica> |
           ( <fórmula> )
```

Las fórmulas se evalúan de acuerdo con los valores de verdad de sus términos constituyentes y las tablas de verdad usuales para los conectores. Surgen los siguientes puntos:

1. Una <*fórmula atómica*> es una expresión con valor de verdad que no involucra conectores y no está contenida entre paréntesis.
2. El símbolo "=►" representa el conector de *implicación lógica.* La expresión/=► *g* está definida para ser lógicamente equivalente a la expresión NOT/OR *g. Nota:* En el capítulo 7 y otros anteriores usamos "IF...THEN..." para este conector.
3. Adoptamos las reglas de precedencia usuales para los conectores (NOT, luego AND, luego OR y luego =►) para reducir la cantidad de paréntesis necesarios para expresar un orden deseado de evaluación.
4. Una **proposición** es simplemente una <*fórmula*>, tal como la definimos anteriormente (usamos el término "fórmula" por consistencia con la siguiente sección).

Reglas de inferencia

Ahora llegamos a las **reglas de inferencia** del cálculo proposicional. Existen muchas de estas reglas. Cada una de ellas es una declaración de la forma

b f => 9

(donde el símbolo |= puede ser leído como **"siempre es el caso que";** observe que necesitamos tales símbolos para poder hacer *metadeclaraciones;* es decir, declaraciones sobre declaraciones). Éstos son algunos ejemplos de reglas de inferencia:

1. **f (f ANO *g*.),<#» *t***
2. h *f* =*■ (*f* OR fir)
3. 1= ((f =► *g*) AND (g =>■ *h*)) =►■ (*f* =*■ *h*)
4. |= (f AND (*f* =*■ *g*)) =*■ *g*

Nota: Ésta es particularmente importante. Se le llama la regla ***modus ponens***. De manera informal, dice que si/es *verdadera* y/implica a g, entonces g también debe ser *verdadera.* Por ejemplo, dado el hecho de que a. y b., a continuación, son *verdaderas*

a. No tengo dinero;

b. Si no tengo dinero tendré que lavar platos;

entonces podemos inferir que c. también es *verdadera:*

c. Tendré que lavar platos.

Para continuar con las reglas de inferencia:

5. (= (f =►■ (*g* =*■ *h*)) = » . ((*f* AND *g*) =*■ *h*)
6. H ((*f* OR *g*) AND (NOT fií OR *h*)) = ► ■ (*f* OR *h*)

Nota: Ésta es otra particularmente importante. Se le llama la regla de **resolución.** Diremos más acerca de ella bajo "Demostraciones" (a continuación) y nuevamente en la sección 23.4.

Demostraciones

Ahora tenemos todo el material necesario para manejar demostraciones formales (en el contexto del cálculo proposicional). El problema de la demostración es el problema de determinar si alguna fórmula *g* dada (la **conclusión)** es consecuencia lógica de algún conjunto dado de fórmulas//,^, ...,*jh* (las **premisas);** en símbolos:

f1, f2, ..., fn* \- *g

(lea *"g* **es deducible a partir** *defl,f2, ...,frí';* observe el uso de otro símbolo metalingüístico, I-). El método básico de proceder es conocido como **encadenamiento hacia adelante.** Este método consiste en aplicar las reglas de inferencia repetidamente en las premisas, en las fórmulas deducidas a partir de esas premisas, en las fórmulas deducidas a partir de esas fórmulas, etcétera; hasta que es posible deducir la conclusión. En otras palabras, el proceso "se encadena hacia adelante" desde las premisas hasta la conclusión. Sin embargo, hay muchas variaciones de este tema básico:

1. **Adopción de una premisa.** Si *g* es de la forma/? =► *q,* adopta a/? como una premisa adi cional y muestra que *q* es deducible a partir de la premisa dada, más /?.
2. **Encadenamiento hacia atrás.** En vez de tratar de demostrar/? =► *q* probamos el **contra positivo NOT** *q* =► **NOT** *p.*
3. **Reducción al absurdo.** En vez de tratar de demostrar/? =► *q* directamente, suponemos que /? y NOT *q* son ambos *verdaderos* y derivamos una contradicción.
4. **Resolución.** Este método usa la regla de inferencia de resolución (número 6 de la lista que dimos anteriormente).

Tratamos la técnica de resolución un poco más a detalle debido a que es de amplia aplicabilidad (en particular, generaliza también el caso del cálculo de predicados, como veremos en la sección 23.4).

Observe primero que la regla de resolución es, en efecto, una regla que nos permite *cancelar subfórmulas;* es decir, dadas las dos fórmulas

f OR *g* *y* NOT *g* OR *h*

podemos cancelar *g* y NOT *g* para derivar la fórmula simplificada

f OR *h*

En particular, si tomamos en cuenta/OR *g* y NOT *g* (es decir, si tomamos a *h* como *verdadero)* podemos derivar a/.

Observe por lo tanto que la regla se aplica generalmente a una *conjunción* (AND) de dos fórmulas, donde cada una de ellas es una *disyunción* (OR) de dos fórmulas. Por lo tanto, para aplicar las reglas de resolución procedemos de la siguiente forma. (Para hacer un poco más concreta nuestra exposición, explicaremos el proceso en términos de un ejemplo específico.) Suponga que queremos determinar si la siguiente demostración supuesta es en realidad válida:

A =»• (*B* =► *C*), NOT *D* OR *A*, *B* \- *D* =+ *C*

(donde *A, B, C y D* son fórmulas). Comenzamos adoptando la negación de la conclusión como una premisa adicional y luego escribimos cada premisa en una línea independiente, de la siguiente manera:

A =*■ { B =f C)*
NOT *D* OR *A*
B
NOT (*D* =► *C)*

A las cuatro líneas les aplicamos "AND" implícitamente.

Ahora convertimos cada línea individual a una **forma normal conjuntiva;** es decir, una forma que consiste en una o más fórmulas unidas con AND y en donde cada fórmula individual contiene (posiblemente) NOTs y Ors, pero no ANDs (vea el capítulo 17). Por supuesto, la segunda y tercera líneas ya están en esta forma. Para convertir las otras dos, primero eliminamos todas las apariciones de "=►" (usando la definición de ese conector en términos de NOT y OR); luego aplicamos las leyes distributivas y las de De Morgan según sea necesario (vea el inicio de esta sección). También eliminamos paréntesis redundantes y pares de NOTs adyacentes (los cuales son cancelados). Las cuatro líneas se convierten en

```
NOT A OR NOT B OR C
NOT D OR A
B
D AND NOT C
```

Luego, a cualquier línea que incluya un AND explícito la reemplazamos por un conjunto de líneas independientes, una por cada una de las fórmulas individuales que tienen AND (eliminando los ANDs en el proceso). En el ejemplo, este paso sólo se aplica a la cuarta línea. Las premisas se ven ahora de esta forma:

```
NOT A OR NOT S OR C
NOT D OR A
S
D
NOT C
```

Ahora podemos comenzar a aplicar la regla de resolución. Seleccionamos un par de líneas que puedan ser *resueltas;* es decir, un par de líneas que contengan (respectivamente) alguna fórmula particular y la negación de esa fórmula. Escojamos las dos primeras líneas que contienen NOT *Ay A,* respectivamente, y resolvámoslas para obtener

```
NOT D OR NOT S OR C
6
D
NOT C
```

(Nota: En general, también necesitamos conservar las dos líneas originales, pero en este ejemplo específico ya no las necesitaremos.) Ahora aplicamos nuevamente la regla, seleccionando otra vez las dos primeras líneas (y resolviendo NOT *B* y *B),* lo cual nos da

```
NOT D OR C
D
NOT C
```

Seleccionamos nuevamente las dos primeras líneas (NOT *D* y *D):*

```
C
NOT C
```

y de nuevo (C y NOT C); el resultado final es el conjunto vacío de proposiciones (representado generalmente por []), lo que interpretamos como una contradicción. Por lo tanto, demostramos el resultado deseado por *reducción al absurdo.*

23.4 CALCULO DE PREDICADOS

Pondremos ahora nuestra atención en el cálculo *de predicados.* La gran diferencia entre el cálculo proposicional y el cálculo de predicados es que este último permite que las fórmulas contengan variables* y cuantificadores, y esto hace que sea mucho más poderoso y que tenga una aplicabilidad más amplia. Por ejemplo, la declaración "el proveedor VI proporciona la parte *p"* y "algún proveedor v proporciona la parte *p"* no son fórmulas válidas para el cálculo proposicional, pero sí son válidas para el cálculo de predicados. Por lo tanto, el cálculo de predicados nos proporciona una base para la expresión de consultas como "¿qué partes proporciona el proveedor VI?" u "obtener los proveedores que proporcionan alguna parte", o incluso "obtener los proveedores que no proporcionan ninguna parte".

Predicados

Como explicamos en el capítulo 3, un **predicado** es una *función con valor de verdad;* es decir, una función que, cuando se le dan los argumentos adecuados para sus parámetros, regresa *verdadero o falso.* Por ejemplo, *">(x,y)"* —escrito más convencionalmente en la forma *"x > y"*— es un predicado con dos parámetros, *x* y *y;* regresa *verdadero* si el argumento que corresponde a *x* es mayor que el argumento que corresponde a *y, y falso* en caso contrario. A un predicado que toma *n* argumentos (es decir, uno que está definido en términos de *n* parámetros) se le llama predicado de *n* lugares. Una proposición (es decir, una fórmula en el sentido de la sección 23.3) puede ser considerada como un predicado de cero lugares; no tiene parámetros y da como resultado *verdadero o falso* inequívocamente.

Es conveniente suponer que los predicados que corresponden a "=", ">", ">", etcétera, están integrados (es decir, son parte del sistema formal que estamos definiendo) y que las expresiones que los usan pueden ser escritas de la manera convencional. Sin embargo, los usuarios también deben tener la posibilidad de definir sus propios predicados (por supuesto). Además, he aquí lo importante: el hecho es —en términos de bases de datos— que un predicado definido por el usuario corresponde a una *varrel* definida por el usuario (como ya sabemos, por lo que dijimos en los capítulos anteriores). Por ejemplo, la varrel V de proveedores puede ser considerada como un predicado de cuatro parámetros, específicamente: V#, PROVEEDOR, STATUS y CIUDAD. Además, las expresiones V (VI, Smith, 20, Londres) y V (V6, White, 45, Roma) representan "ejemplares", "instanciaciones" o "invocaciones" de ese predicado que (si tomamos nuestro conjunto de valores de ejemplo usuales) dan como resultado *verdadero y falso,* respectivamente. De manera informal, podemos considerar tales predicados —junto con cualquier *restricción de integridad* aplicable, que también es predicado— como la definición de lo que "significa" la base de datos, como explicamos en partes anteriores de este libro (en particular en el capítulo 8).

Fórmulas bien formadas

El siguiente paso es ampliar la definición de "fórmula". Para evitar confusión con las fórmulas de la sección anterior (que son de hecho un caso especial), ahora cambiaremos al término **fórmula**

---

*Las variables en cuestión son variables *lógicas* y no de lenguaje de programación. Para nuestros propósitos podemos pensar en ellas como variables de alcance en el sentido del capítulo 7.

bien formada (WFF, pronunciado "weff') del capítulo 7. Ésta es una sintaxis simplificada para las WFFs:

```
<wff>       | <término> NOT ( <wff> ) ( <wff> ) AND
            | ( <wff> ) ( <wff> ) OR ( <wff> ) (
            | <wff> ) =► ( <wff> ) EXISTS <nombre
            | de variable> ( <wff> ) FORALL <nombre
            | de variable> ( <wff> )

<término>  ::= [ NOT ] <nombre de predicado> [ <lista de argumentos separados
                                                con comas> ) ]
```

Surgen estos puntos:

1. Un <*término*> es simplemente un "ejemplar de predicado" posiblemente negado (si pensa mos en un predicado como una función con valor de verdad, entonces un ejemplar de predi cado es una llamada a esa función). Cada <*argumento*> debe ser una constante, un nombre de variable o una invocación a función, donde cada argumento de una invocación a función es a su vez una constante o nombre de variable o invocación a función. La <*lista de argu mentos separados con comas*> y (opcionalmente) los paréntesis correspondientes, son omi tidos para un predicado de cero lugares. *Nota:* Las funciones (es decir, sobre y arriba de las funciones con valor de verdad que son los predicados) son permitidas para hacer posible que las WFFs incluyan expresiones computacionales como *"+(x,y)"* —escrito más convencionalmente "x + *y"*—, etcétera.
2. Al igual que en la sección 23.3, adoptamos las reglas de precedencia usuales para los conectores (NOT, luego AND, luego OR, luego = ►) para reducir la cantidad de paréntesis necesarios para expresar un orden de evaluación deseado.
3. Suponemos que usted está familiarizado con los cuantificadores EXISTS *y* FORALL. *Nota:* Aquí sólo estamos preocupados por el cálculo de predicados de *primer orden,* lo cual significa básicamente que (a) no hay "variables de predicado" (es decir, variables cuyos valores permitidos son predicados) y por lo tanto, que (b) los propios predicados no pueden estar sujetos a una cuantificación. Vea el ejercicio 7.9 en el capítulo 7.
4. Las leyes de De Morgan pueden generalizarse para ser aplicadas a WFFs cuantificadas de la siguiente forma:

```
NOT ( FORALL X (     )        EXISTS X ( NOT ( f ) )
NOT ( EXISTS x (    ) )       FORALL x ( NOT ( f) )
                    )
```

Este punto también lo explicamos en el capítulo 7.

5. Para repetir otro punto más del capítulo 7: Dentro de una WFF dada, cada referencia a una variable es **libre** o **ligada.** Una referencia es ligada si (a) aparece inmediatamente después de un cuantificador (es decir, indica la *variable cuantificada)* o (b) se encuentra dentro del alcance de un cuantificador y hace referencia a la variable cuantificada aplicable. Una referencia a variable es libre si y sólo si no está ligada.
6. Una **WFF cerrada** es aquella que no contiene referencias a variables libres (de hecho, es una proposición). Una **WFF abierta** es aquella que no está cerrada.

Interpretaciones y modelos

¿Qué *significan* las WFFs? Para proporcionar una respuesta formal a esta pregunta presentamos la noción de una **interpretación.** Una interpretación de un conjunto de WFFs se define de la siguiente manera:

- Primero especificamos un **universo de discurso** sobre el cual vamos a interpretar esas WFFs. En otras palabras, especificamos una *correspondencia* entre (a) las constantes permitidas del sistema formal (los valores de dominio, en términos de base de datos) y (b) los objetos de la "realidad". Cada constante individual corresponde precisamente a un objeto en el universo de discurso.
- Segundo, especificamos un significado para cada predicado en términos de los objetos que están en el universo de discurso.
- Tercero, también especificamos un significado para cada función en términos de los obje tos que están en el universo de discurso.

Entonces, la interpretación consiste en la combinación del universo de discurso más la correspondencia de las constantes individuales con los objetos de ese universo y además, los significados definidos para los predicados y funciones con respecto a ese universo.

A manera de ejemplo, sea el universo de discurso el conjunto de enteros {0, 1, 2, 3, 4, 5}; sea que constantes como "2" correspondan a objetos de ese universo en la forma obvia y sea que el predicado *"x > y"* esté definido para que tenga el significado usual. (También podríamos definir funciones como "+", "-", etcétera, si fuera necesario.) Ahora podemos asignar valores de verdad a WFFs como las siguientes, de la forma en que indicamos:

```
2 > 1                    : verdadero
2 > 3                    : falso
EXISTS  x ( x > 2 )      : verdadero
FORALL  x ( x > 2 )      : falso
```

Observe, sin embargo, que son posibles otras interpretaciones. Por ejemplo, podríamos especificar que el universo de discurso sea un conjunto de niveles de clasificación de seguridad como los siguientes:

```
destruir antes de leer    (nivel 5)
destruir después de leer  (nivel 4)
muy secreto               (nivel 3)
secreto                   (nivel 2)
confidencial              (nivel 1)
no clasificado            (nivel 0)
```

El predicado ">" podría significar ahora "más seguro que" (es decir, clasificación más alta).

Ahora, probablemente se habrá dado cuenta de que las dos interpretaciones posibles son *isomórficas;* es decir, es posible especificar una correspondencia de uno a uno entre ellas y por lo tanto, en un nivel más profundo las dos interpretaciones en realidad son una y de hecho la misma. Pero debe comprender claramente que pueden existir interpretaciones que sean genuinamente diferentes en tipo. Por ejemplo, podríamos tomar nuevamente el universo de discurso como los enteros del 0 al 5, pero definir el predicado ">" para que significara *igualdad.* (Por supuesto, causaríamos probablemente mucha confusión en esta forma, pero al menos estaríamos dentro de nuestro derecho de hacerlo.) Ahora la primera WFF de la lista anterior daría como resultado *falso* en lugar de *verdadero.*

Otro punto que debe comprender claramente es que dos interpretaciones pueden ser genuinamente diferentes en el sentido anterior y aún así dar los mismos valores de verdad para el conjunto dado de WFFs. Éste sena el caso con las dos definiciones diferentes de ">" en nuestro ejemplo, si omitiéramos la WFF "2 >1".

Observe, de paso, que todas las WFFs que hemos tratado hasta el momento en esta subsección han sido WFFs *cerradas.* La razón es que, dada una interpretación, siempre es posible asignar un valor de verdad sin ambigüedad a una WFF cerrada, pero el valor de verdad de una WFF abierta dependerá de los valores asignados a las variables libres. Por ejemplo, la WFF abierta

```
x > 3
```

es (obviamente) *verdadera* si el valor de *x* es mayor que 3 *y falsa* en caso contrario (sin importar qué signifique "mayor que" y "3" en la interpretación).

Ahora definimos un **modelo** de un conjunto de WFFs (necesariamente cerradas) para que sea una interpretación para la cual todas las WFFs del conjunto sean *verdaderas.* Las dos interpretaciones dadas anteriormente para las cuatro WFFs

```
2 > 1
2 > 3
EXISTS x ( x > 2 )
FORALL X ( X > 2 )
```

en términos de los enteros 0 a 5 no fueron modelos para esas WFFs, debido a que algunas de las mismas dan como resultado *falso* bajo esa interpretación. Por el contrario, la primera interpretación (en la cual ">" fue definido "adecuadamente") *habría* sido un modelo para el conjunto de WFFs

```
2 > 1
3 > 2
EXISTS X ( X > 2 )
FORALL x ( x > 2 OR NOT ( x > 2 ) )
```

Por último, observe que debido a que un conjunto dado de WFFs puede admitir varias interpretaciones en las cuales todas las WFFs dan como resultado *verdadero,* este conjunto puede tener por lo tanto varios *modelos* (en general). Así, una base de datos puede tener varios modelos (en general) debido a que —desde la perspectiva de la teoría de modelos— una base de datos *es* simplemente un conjunto de WFFs. Vea la sección 23.5.

Forma clausal

Así como cualquier fórmula del cálculo proposicional puede ser convertida a la forma normal conjuntiva, cualquier WFF del cálculo de predicados puede ser convertida a una **forma clausal,** que puede ser considerada como una versión extendida de la forma normal conjuntiva. Una motivación para hacer tal conversión es que (de nuevo) nos permite aplicar la regla de resolución en la construcción o verificación de demostraciones, como veremos.

El proceso de conversión es realizado de la manera siguiente (a grandes rasgos; para mayores detalles vea la referencia [23.10]). Ilustraremos los pasos aplicándolos a una WFF de ejemplo que es la siguiente

```
FORALL x ( p ( x ) AND EXISTS y ( FORALL z ( q ( y, z ) ) ) )
```

Aquí *p* y *q* son predicados, y *x, y* y *z* son variables.

1. Eliminar los símbolos "=►" como en la sección 23.3. En nuestro ejemplo, esta primera transformación no tiene efecto alguno.
2. Usar las leyes de De Morgan —más el hecho de que los NOTs adyacentes se cancelan— para mover los NOTs a fin de que sólo tengan efecto sobre los términos y no sobre las WFFs generales. (De nuevo, esta transformación en particular no tiene efecto en nuestro ejemplo específico.)
3. Convertir la WFF a la **forma normal prenexa** moviendo todos los cuantificadores a la parte inicial (renombrando sistemáticamente a las variables, en caso necesario):

   ```
   FORALL x ( EXISTS y ( FORALL z ( p ( x ) AND q ( y, z ) ) ) )
   ```

4. Observe que una WFF cuantificada existencialmente como

   ```
   EXISTS v ( r ( v ) )
   ```

   es equivalente a la WFF

   ```
   r ( a )
   ```

   para alguna constante *a* desconocida; es decir, la WFF original asevera que dicha *a* existe aunque simplemente no sabemos su valor. De manera similar, una WFF como

   ```
   FORALL u ( EXISTS v ( s ( u, v ) ) )
   ```

   es equivalente a la WFF

   ```
   FORALL u ( s ( u, f ( u ) ) )
   ```

   para alguna función/desconocida de la variable *u* cuantificada universalmente. La constante a y la función/de estos ejemplos son conocidas, respectivamente, como **constante de Skolem y función de Skolem,** en honor al lógico T. A. Skolem. *(Nota:* Una constante de Skolem es en realidad sólo una función de Skolem sin argumentos.) Entonces, el siguiente paso es eliminar los cuantificadores existenciales, reemplazando las variables cuantificadas correspondientes por funciones de Skolem (cualesquiera) de todas las variables cuantificadas universalmente que preceden al cuantificador en cuestión en la WFF:

   ```
   FORALL X ( FORALL z ( p ( X ) AND q ( f ( X ), z ) ) )
   ```

5. Ahora todas las variables están cuantificadas universalmente. Por lo tanto, podemos adop tar una convención mediante la cual todas las variables estén cuantificadas universalmente *en forma implícita* y eliminar así los cuantificadores explícitos:

   ```
   p ( x ) AND q ( f ( x ), z )
   ```

6. Convertir la WFF a la forma normal conjuntiva; es decir, a un conjunto de cláusulas unidas todas por ANDs (donde cada cláusula involucra posiblemente NOTs y ORs, pero no ANDs). En nuestro ejemplo, la WFF ya está en esta forma.
7. Escribir cada cláusula en una línea independiente y eliminar los ANDs:

```
P i  X )
q (  f ( x ),  z )
```

Ésta es la forma clausal equivalente de la WFF original.

Nota: Del procedimiento anterior se desprende que la forma general de una WFF en forma clausal es un conjunto de cláusulas, cada una en su línea y cada una de la forma:

```
NOT A1 OR NOT A2 OR      OR NOT Am OR B1 OR B2 OR      OR Bn
```

donde todas las *A* y *B* son términos no negados. Podemos volver a escribir una cláusula de éstas, si lo deseamos, como:

```
A1 AND A2 AND      AND Am =*■ B1 OR B2 OR      OR Bn
```

Si hay a lo mucho una *B (n* = 0 o 1), a la cláusula se le llama **cláusula de Horn** en honor al lógico Alfred Horn.

Uso de la regla de resolución

Ahora estamos en posición para ver la forma en que un sistema de base de datos basado en la lógica puede manejar consultas. Usamos el ejemplo que está al final de la sección 23.2. Primero tenemos un predicado MADRE_DE, el cual involucra dos parámetros que representan a la madre y a la hija, respectivamente, y se nos han dado los dos siguientes términos (ejemplares de predicado):

```
1. MADRE_DE ( Ana, Bety )
2. MADRE_DE ( Bety, Celia )
```

También se nos ha dado la siguiente WFF (el "axioma deductivo"):

```
3. MADRE_DE  ( x, y ) AND MADREJ5E  ( y,  z )  =►
                                    ABUELAJ3E  (  X,  z )
```

(observe que ésta es una cláusula de Horn). Para simplificar la aplicación de la regla de resolución, volvamos a escribir la cláusula para eliminar el símbolo "=►":

```
4. NOT MADRE DE    x, y ) OR NOT MADRE_DE ( y, z )
                                          OR ABUELA_DE ( x, z )
```

Continuamos para demostrar que Ana es la abuela de Celia —es decir, vamos a mostrar la manera de responder la pregunta "¿es Ana la abuela de Celia?". Comenzamos negando la conclusión que vamos a demostrar y la adoptamos como premisa adicional:

```
5. NOT ABUELA_DE ( Ana, Celia )
```

Ahora, para aplicar la regla de resolución, debemos sustituir sistemáticamente los valores por las variables en forma tal que podamos encontrar dos cláusulas que contengan, respectivamente, una WFF y su negación. Tal sustitución es legítima, ya que todas las variables están cuantificadas umversalmente en forma implícita y por lo tanto, las WFFs individuales (no negadas) deben ser *verdaderas* para todas y cada una de las combinaciones válidas de valores de sus variables. *Nota:* Al proceso de encontrar un conjunto de sustituciones que hagan que estas dos cláusulas sean resolubles de esta manera, se le conoce como **unificación.**

Para ver la forma en que funciona lo anterior, en el caso que tenemos, observe primero que las líneas 4 y 5 contienen los términos ABUELA_DE *(x, z)* y NOT ABUELA_DE (Ana, Celia), respectivamente. Por lo tanto, sustituimos Ana en lugar de *x* y Celia en lugar de *z* en la línea 4 y resolvemos para obtener:

```
6. NOT MADREDE (Ana, y ) OR NOT MADRE_DE ( y, Celia )
```

La línea 2 contiene MADRE__DE (Bety, Celia). Por lo tanto, sustituimos Bety en lugar de *y* en la línea 6 y resolvemos para obtener:

```
7. NOT MADREDE ( Ana, Bety )
```

Al resolver la línea 7 y la línea 1 obtenemos el conjunto vacío de cláusulas []: *contradicción.* Por lo tanto, la respuesta a la consulta original es "sí, Ana es la abuela de Celia".

Qué hay acerca de la consulta "¿quiénes son las nietas de Ana?". Observe primero que el sistema no sabe nada acerca de nietas, sino que sólo sabe acerca de abuelas. Podríamos añadir otro axioma deductivo para decir que *z* es la nieta de *x* si y sólo si *x* es la abuela de *z* (no se permiten hombres en esta base de datos). Por supuesto, en forma alterna podríamos cambiar la redacción de la pregunta a "¿de quién es abuela Ana?". Consideremos esta última formulación. Las premisas son (para repetir):

```
1. MADRE_DE ( Ana,  Bety )

2. MADREDE ( Bety, Celia )

3. NOT MADREDE ( x, y ) OR NOT MADRE_DE ( y, z )
                                        OR ABUELA_DE ( X, z )
```

Introducimos una cuarta premisa que es la siguiente:

```
4. NOT ABUELADE ( Ana, r ) OR RESULTADO ( r )
```

En forma intuitiva, esta nueva premisa establece que Ana no es la abuela de nadie o, en forma alterna, que hay alguna persona *r* que pertenece al resultado (debido a que Ana *es* la abuela de esa persona r). Queremos descubrir la identidad de todas esas personas *r.* Procedemos de la siguiente manera:

Primero sustituimos Ana en lugar de *x* y *r* en lugar de z, y resolvemos las líneas 4 y 3 para obtener:

```
5. NOT MADRE_DE ( Ana, y ) OR NOT MADREDE { y, z )
                                          OR RESULTADO ( z )
```

Luego sustituimos Bety en lugar de *y,* y resolvemos las líneas 5 y 1 para obtener:

```
6. NOT MADRE_DE ( Bety, z ) OR RESULTADO ( z )
```

Ahora sustituimos Celia en lugar de *z,* y resolvemos las líneas 6 y 2 para obtener:

```
7. RESULTADO ( Celia )
```

Por lo tanto, Ana es la abuela de Celia.

Nota: Si nos hubieran dado un término adicional como el siguiente:

```
MADREDE ( Bety, Delia )
```

entonces podríamos haber sustituido Delia en lugar de *z* en el paso final (en lugar de Celia) y obtener:

```
RESULTADO ( Delia )
```

Por supuesto, los usuarios esperan ver ambos nombres en el resultado. Por lo tanto, el sistema necesita aplicar exhaustivamente el proceso de unificación y resolución para generar todos los valores de resultado *posibles.* Los detalles de este refinamiento están fuera del alcance de la presente explicación.

23.5 LAS BASES DE DATOS DESDE LA PERSPECTIVA DE LA TEORÍA DE DEMOSTRACIONES

Como explicamos en la sección 23.4, una *cláusula* es una expresión de la forma:

A1 AND *A2* AND . . . AND *Am* =*■ B1* OR *B2* OR ... OR *Bn*

donde las *A* y *B* son términos de la forma:

r (x1, x2, ..., xt)

(aquí *r* es un predicado y *xl, x2, ...,xt* son los argumentos para ese predicado). Para seguir la referencia [23.12], ahora consideramos dos casos especiales importantes de esta construcción general:

- *Caso 1: m* = 0, *n* = 1

  En este caso la cláusula puede ser simplificada a sólo:

  =▶• *B1*

  o en otras palabras (si eliminamos el símbolo de implicación), a

  r (x1, x2, ..., xt)

  para algún predicado *r* y algún conjunto de argumentos *xl, x2,..., xt.* Si todas las *x* son constantes, la cláusula representa un **axioma base,** es decir, es una declaración que es inequívocamente *verdadera.* En términos de bases de datos, dicha declaración corresponde a una tupia de alguna varrel *R** El predicado *r* corresponde al "significado" de la varrel *R.* como explicamos en muchas partes de este libro. Por ejemplo, en la base de datos de proveedores y partes hay una varrel llamada VP, cuyo significado es que el proveedor indicado (V#) proporciona la parte indicada (P#) en la cantidad indicada (CANT). Observe que este significado corresponde a una **WFF abierta,** ya que incluye referencias a variables libres (V#, P# y CANT). Por el contrario, la tupia (VI, Pl, 300) —en la cual todos los argumentos son constantes— es un axioma base, o **WFF cerrada,** que asevera inequívocamente que el proveedor VI proporciona la parte Pl en una cantidad de 300.

- *Caso 2: m* > 0, *n* = 1

  En este caso la cláusula toma la forma

  A1 AND *A2* AND ... AND *Am* =»

---

*O a un valor en algún dominio.

la cual puede ser considerada como un **axioma deductivo;** da una definición (tal vez incompleta) del predicado que está al lado derecho del símbolo de implicación, en términos de los predicados que están al lado izquierdo (para un ejemplo, vea la definición anterior del predicado ABUELA_DE).

En forma alterna, una cláusula de éstas puede ser considerada como la definición de una **restricción de integridad** (o una restricción de *varrel,* como sucede usando la terminología del capítulo 8). Suponga, para efectos del ejemplo, que la varrel V de proveedores tiene sólo dos atributos: V# y CIUDAD. Entonces la cláusula

```
V ( v, C1 ) AND V ( v, c2 ) =► C1 = C2
```

expresa la restricción de que CIUDAD es funcionalmente dependiente de V#. Observe en este ejemplo el uso del predicado integrado "=".

Como lo demuestra la explicación anterior, las tupias en relaciones ("axiomas base"), las relaciones derivadas ("axiomas deductivos") y las restricciones de integridad pueden ser consideradas como casos especiales de la construcción de *cláusula* general. Tratemos ahora de ver la manera en que estas ideas pueden conducir a la vista de una base de datos "por la teoría de demostraciones" que mencionamos en la sección 23.2.

Primero, la vista tradicional de una base de datos puede ser considerada como una teoría de **modelos.** Aquí por "vista tradicional" simplemente queremos decir una vista en la cual una base de datos se percibe como consistente en una colección de varrels explícitamente nombradas, y donde cada una consiste en un conjunto de tupias explícitas, junto con un conjunto explícito de restricciones de integridad. Ésta es la percepción que puede ser caracterizada como una **teoría de modelos,** como ahora explicaremos.

- Los dominios subyacentes contienen valores o constantes que tienen supuestamente un signifi cado para determinados objetos de la "realidad" (de forma más precisa, para alguna **interpreta ción,** en el sentido de la sección 23.4). Por lo tanto, corresponden con el universo de discurso.
- Las varrels (de forma más precisa, los *encabezados* de varrel) representan un conjunto de predicados, o WFFs abiertas, que van a ser interpretados sobre ese universo. Por ejemplo, el encabezado de la varrel VP representa el predicado "el proveedor V# proporciona la parte P# en la cantidad CANT".
- Cada tupia en una varrel dada representa un ejemplar del predicado correspondiente; es decir, representa una proposición (una WFF cerrada; no contiene variables) que es inequí vocamente *verdadera* en el universo de discurso.
- Las restricciones de integridad también son WFFs cerradas y son interpretadas sobre el mismo universo. Debido a que los datos no violan las restricciones (es decir, ¡no *deben]),* estas restricciones también dan necesariamente como resultado *verdadero.*
- Las tupias y las restricciones de integridad pueden ser consideradas como el conjunto de axio mas que definen una determinada **teoría lógica** (en términos generales, en la lógica una "teoría" *es* un conjunto de axiomas). Puesto que todos estos axiomas son *verdaderos* en la in terpretación, entonces (por definición) esa interpretación es un **modelo** de esa teoría lógica en el sentido de la sección 23.4. Observe que, como señalamos en esa sección, es posible que el modelo no sea único; es decir, una base de datos dada puede tener varias interpretaciones posi bles, donde todas ellas son igualmente válidas desde un punto de vista lógico.

Por lo tanto, desde la perspectiva de la teoría de modelos, el "significado" de la base de datos *es* el modelo, en el sentido anterior del término "modelo". Y debido a que hay muchos

modelos posibles, hay muchos significados posibles (al menos en principio).* Además, el procesamiento de consultas desde la perspectiva de la teoría de modelos, es esencialmente un proceso de evaluación de una determinada WFF abierta para descubrir qué valores de las variables libres de esa WFF ocasionan que la WFF dé como resultado *verdadero* dentro del modelo.

Es suficiente sobre la perspectiva de la teoría de modelos. Sin embargo, para poder aplicar las reglas de inferencia que describimos en las secciones 23.3 y 23.4, es necesario adoptar una perspectiva diferente: una en la cual la base de datos sea considerada explícitamente como una teona lógica determinada; es decir, como un conjunto de axiomas. Entonces, el "significado" de la base de datos se convierte precisamente en la colección de todas las declaraciones *verdaderas* que pueden ser deducidas a partir de esos axiomas; es decir, es el conjunto de **teoremas** que pueden ser demostrados a partir de esos axiomas. Ésta es la perspectiva de la **teoría de demostraciones.** En esta perspectiva, la evaluación de la consulta se convierte en un proceso de demostración de teoremas (en términos conceptuales, a cualquier velocidad; aunque es probable que por eficiencia, el sistema use técnicas más convencionales de procesamiento de consultas, como veremos en la sección 23.7)

Nota: Del párrafo anterior se desprende que una diferencia entre la perspectiva de la teoría de modelos y la de demostraciones (hablando en forma intuitiva) es que mientras que una base de datos puede tener "muchos significados" desde la perspectiva de la teoría de modelos, por lo general tiene precisamente un solo "significado" desde la teoría de demostraciones; salvo que como señalamos anteriormente, (a) ese único significado es en realidad *el* significado canónico en el caso de la teoría de modelos, y en cualquier caso (b) la indicación de que sólo hay un significado en el caso de la teoría de demostraciones deja de ser verdadera —en general— cuando la base de datos incluye cualquier axioma negativo [23.9] y [23.10].

Los axiomas para una base de datos dada (desde la perspectiva de la teoría de demostraciones) pueden ser resumidos informalmente de la siguiente manera [23.15]:

1. Los axiomas base que corresponden a los valores en los dominios y a las tupias en las varrels base. Estos axiomas constituyen lo que a veces se denomina como la **base de datos extensional** (en oposición a la base de datos *intensional;* vea la siguiente sección).
2. Un "axioma de conclusión" para cada varrel, el cual establece que la falla de una tupia —que por lo demás sería válida— para que aparezca en la varrel en cuestión, puede ser interpre tada como si significara que la proposición que corresponde a esa tupia *es falsa.* (De hecho, es claro que estos axiomas de conclusión, en conjunto, constituyen la **Suposición de mundo cerrado** que ya tratamos en el capítulo 7.) Por ejemplo, el hecho de que la varrel V de pro veedores no incluya la tupia (V6, White, 45, Roma) significa que la proposición "existe un proveedor V6 llamado White con status 45 y ubicado en Roma" *es falsa.*
3. El axioma de "nombre único", que establece que toda constante es distinguible con respecto a todas las demás (es decir, tiene un nombre único).
4. El axioma de "cierre de dominio", que establece que no existen más constantes que aquellas que están en los dominios de la base de datos.
5. Un conjunto de axiomas (en esencia, estándar) para definir el predicado de *igualdad* inte grado. Estos axiomas son necesarios debido a que los axiomas que están en los números 2, 3 y 4 anteriores utilizan el predicado de igualdad.

*Sin embargo, si damos por hecho que la base de datos no contiene explícitamente ninguna información negativa (por ejemplo, una proposición de la forma "NOT V#(V9)", la cual significa que V9 no es un número de proveedor), también existirá un significado "mínimo" o *canónico* que es la intersección de lodos los modelos posibles [23.10]. Además, en este caso ese significado canónico será el mismo que el adscrito a la base de datos desde la perspectiva de la teoría de demostraciones, que explicaremos en un momento.

Concluimos esta sección con un breve resumen de las diferencias principales entre las dos perspectivas (teoría de modelos y teoría de demostraciones). En primer lugar, ¡hay que decir que, desde un punto de vista completamente pragmático, es posible que no haya mucha diferencia!, al menos en los términos de los DBMSs actuales. Sin embargo:

- Los números 2 a 5 en la lista de axiomas para la perspectiva de la teoría de demostraciones, hacen determinadas suposiciones explícitas implicadas en la noción de interpretación de la perspectiva de la teoría de modelos [23.15]. Por lo general, el establecimiento explícito de suposiciones es una buena idea; además, es necesario especificar explícitamente esos axiomas adicionales para poder aplicar técnicas de demostración generales, como el método de resolución que describimos en las secciones 23.3 y 23.4.
- Observe que la lista de axiomas no hace mención de las restricciones de integridad. La razón de esta omisión es que (desde la perspectiva de la teoría de demostraciones) la incorporación de tales restricciones convierte al sistema en un DBMS **deductivo.** Vea la sección 23.6.
- La perspectiva de la teoría de demostraciones goza de una cierta elegancia que no tiene la pers pectiva de la teoría de modelos, en cuanto a que proporciona una percepción uniforme de varias construcciones que por lo general, son vistas como si fueran más o menos distintas: datos base, consultas, restricciones de integridad (no obstante el punto anterior), datos virtuales, etcétera. Por consecuencia, surge la posibilidad de tener interfaces e implementaciones más uniformes.
- La perspectiva de la teoría de demostraciones también proporciona una base natural para el tratamiento de ciertos problemas que tradicionalmente han sido siempre difíciles para los sistemas relaciónales —como la **información disyuntiva** (por ejemplo, "el proveedor V6 está ubicado en París o en Roma"), la derivación de **información negativa** (por ejemplo, "¿quién no es un proveedor?") y las **consultas recursivas** (vea la siguiente sección)— aunque en este último caso, por lo menos, no hay razón por la cual un sistema relacional clásico no pueda en principio ser extendido adecuadamente para manejar tales consultas, y varios productos comerciales ya lo han hecho (vea también el apéndice B). En las secciones 23.6 y 23.7, tendremos más que decir con respecto a estos aspectos.
- Por último, para citar a Reiter [23.15], la perspectiva por la teoría de demostraciones "pro porciona un tratamiento correcto de [extensiones para] el modelo relacional que incorpora más semántica de la realidad" (como dijimos en la sección 23.2).

23.6 SISTEMAS DE BASES DE DATOS DEDUCTIVAS

Un **DBMS deductivo** es aquel que soporta la perspectiva de una base de datos por la teoría de demostraciones y —en particular— es capaz de deducir o inferir hechos adicionales a partir de los hechos dados en la base de datos extensional (aplicando **axiomas deductivos** o **reglas de inferencia** especificados para esos hechos dados).* Los axiomas deductivos, junto con las restricciones de integridad (que tratamos más adelante), forman lo que en ocasiones se conoce como **base de datos intensional;** y la base de datos extensional y la intensional juntas constituyen lo que generalmente se conoce como *base de datos deductiva* (aunque no es un buen término, debido a que es el DBMS, y no la base de datos, quien realiza las deducciones).

*Con respecto a esto, vale la pena observar que Codd decía desde 1974 que uno de los objetivos del modelo relacional era, precisamente, "combinar los campos de la recuperación de hechos y la administración de archivos preparándose para la adición, en un tiempo posterior, de servicios inferenciales en el mundo comercial" [11.2], [25.8].

Como acabamos de indicar, los axiomas deductivos forman una parte de la base de datos intensional. La otra parte consiste en axiomas adicionales que representan restricciones de integridad (es decir, las reglas cuyo propósito principal es restringir las actualizaciones; aunque de hecho, dichas reglas también pueden ser usadas en el proceso de deducir hechos adicionales a partir de los que se han dado).

Veamos cómo se vena la base de datos de partes y proveedores de la figura 3.8 en forma de "DBMS deductivo". Primero, habrá un conjunto de axiomas base que definan los valores de dominio válidos. *Nota:* En lo que viene a continuación, por razones de legibilidad adoptamos esencialmente las mismas convenciones con respecto a la representación de valores como lo hicimos (por ejemplo) en la figura 3.8; por lo tanto, escribimos 300 como una abreviatura adecuada para CANT(300), etcétera.

```
V#  ( V1     NOMBRE ( Smith        STATUS   ( 5 )    CIUDAD     ( Londres )
      )      NOMBRE ( Jones        STATUS   ( 10 )   CIUDAD     ( Paris (
V#  ( V2     NOMBRE ( Blake        STATUS   ( 15 )   CIUDAD     Roma (
      )      NOMBRE ( Clark        etcétera          CIUDAD     Atenas
V#  ( V3     NOMBRE ( Adams                          etcétera
      )
V# ( V4
etcétera   NOMBRE

                                )

                                )

                                )

                       ( White
                       ( Tuerca

                NOMBRE
                NOMBRE
                                )
                                )

                                )

                                )
                NOMBRE ( Tornillo )
                etcétera
```

etcétera.

Luego, habrá axiomas base para las tupias en las relaciones base:

```
V ( V1, Smith, 20, Londres )
V ( V2, jones, 10, París  )
etcétera

P ( P1, Tuerca, Rojo, 12, Londres )
etcétera

VP ( V1, P1, 300 )
etcétera
```

Nota: No estamos sugiriendo seriamente que la base de datos extensional será creada listando explícitamente todos los axiomas base como acabamos de indicar, sino que (por supuesto) se usarán los métodos tradicionales de definición y captura de datos. En otras palabras, los DBMSs deductivos aplicarán, por lo general, sus deducciones para bases de datos convencionales que ya existan y que hayan sido construidas en la forma convencional. Sin embargo, ¡observe que ahora se vuelve más importante que nunca que la base de datos extensional no viole ninguna de las restricciones de integridad declaradas!; ya que una base de datos que viola cualquiera de estas restricciones, representa (en términos lógicos) un conjunto de axiomas inconsistentes, y es bien sabido que, partiendo de ese punto, es posible demostrar que es "verdadera" *absolutamente cualquier proposición que se quiera* (en otras palabras, es posible derivar contradicciones). Por la misma razón, también es importante que el conjunto de restricciones de integridad establecido sea consistente.

Ahora para la base de datos intensional. Éstas son las restricciones de atributos:

```
V ( v, vn, vt, ve ) ➡ V# ( v ) AND
                      NOMBRE ( vn ) AND
                      STATUS ( vt ) AND
```

CIUDAD (*ve*)

```
p ( p, pn, pl, pp, pe ) =<  P# ( p ) AND
                            NOMBRE ( pn ) AND
                            COLOR ( pl ) AND
                            PESO ( pp ) AND
                            CIUDAD ( pe )
etcétera
```

Restricciones de clave candidata:

```
V ( v, vn1, vt1, vd    AND V ( V, vn2, Vt2,  vc2 )
                       =►• vn1 = vn2 AND
                           vt1 • vt2 AND
                           vd = vc2
etcétera
```

Restricciones de clave externa:

```
P, C ) =► V       vn, vt, ve ) AND
                  pn, pl, pp, pe )
```

Y así sucesivamente. *Nota:* Para efectos de la explicación, suponemos que las variables que aparecen al lado derecho del símbolo de implicación y que no aparecen al lado izquierdo (en el ejemplo, *vn, vt,* etcétera) están cuantificadas existencialmente. (Todas las demás están cuantificadas universalmente, como explicamos en la sección 23.4.) Técnicamente, necesitamos algunas funciones de Skolem; *vn,* por ejemplo, en realidad debería ser reemplazada por —digamos— VN (v), donde VN es una función de Skolem.

Observe de paso que la mayoría de las restricciones que mostramos anteriormente no son cláusulas puras en el sentido de la sección 23.5, ya que el lado derecho no es simplemente una disyunción de términos simples.

Añadamos ahora algunos axiomas deductivos adicionales:

```
V ( v, vn, vt, ve ) AND vt > 15
                    = ► BUENPROVEEDOR ( v, vt, ve )
```

(compárelo con la definición de la vista BUEN_PROVEEDOR del capítulo 9, sección 9.1).

```
V ( vx, vxn, vxt, ve ) AND V ( vy, vyn, vyt, ve )
                                  =»■ VV_COUBICADO ( vx, vy )
V ( v, vn, vt, c ) AND P ( p, pn, pl, pp, c )
                                  = + VP_C0UBICAD0 ( v, p )
```

y así sucesivamente.

Para hacer el ejemplo un poco más interesante, ampliemos ahora la base de datos para que incluya una varrel de "estructura de partes", que muestra qué partes *px* contienen qué partes *py* como componentes inmediatos (es decir, de primer nivel). Primero, una restricción para mostrar que *px* y *py* deben identificar partes existentes:

```
ESTRUCTURAPARTES ( px, py )  => ? ( px, xn, xl,      xc ) AND
                                                     ye )
                                    P ( py, yn, yi,
```

Algunos valores de datos:

```
ESTRUCTURA_PARTES ( P1, P2 )
ESTRUCTURA_PARTES ( P1, P3 )
ESTRUCTURAPARTES ( P2, P3 )
ESTRUCTURAPARTES ( P2, P4 )
etcétera
```

(En la práctica, es probable que ESTRUCTURA_PARTES también tenga un argumento de "cantidad" que muestre cuántas *py* son necesarias para hacer una *px*, pero omitimos este refinamiento por razones de simplicidad.)

Ahora añadimos un par de axiomas deductivos para explicar lo que significa que la parte *px* contenga a la parte *py* como componente (a cualquier nivel):

```
ESTRUCTURAPARTES ( px, py ) => COMPONENTE_DE ( px, py )

ESTRUCTURA_PARTES ( px, pz ) AND COMPONENTEDE ( pz, py )
                             = *■ COMPONENTEDE ( px, py )
```

En otras palabras, la parte *py* es un componente de la parte *px* (a algún nivel) si es un componente inmediato de la parte *px* o un componente inmediato de alguna parte *pz* que a su vez es componente (a algún nivel) de la parte *px*. Observe que el segundo axioma es recursivo; define al predicado COMPONENTE_DE en términos de sí mismo.* Por el contrario, los sistemas relacionales no han permitido, históricamente, que las definiciones de vistas (o las consultas, o las restricciones de integridad, etcétera) sean recursivas de esta forma. Esta habilidad para soportar la recursion es una de las distinciones más inmediatamente obvias entre los DBMSs deductivos y sus contrapartes relacionales clásicas; aunque como mencionamos en la sección 23.5 (y como vimos en el capítulo 6), no existe una razón fundamental de por qué los sistemas relacionales clásicos no deban ser extendidos para soportar tal recursion, y en algunos ya se ha hecho. Tendremos más que decir con respecto a la recursion en la sección 23.7.

Datalog

De lo que explicamos anteriormente, debe quedar claro que una de las partes más directamente visibles de un DBMS deductivo será un lenguaje en el cual se formulen los axiomas deductivos (llamados **reglas,** por lo general). El ejemplo más conocido de uno de estos lenguajes se llama **Datalog** (por analogía con Prolog) [23.9]. En esta subsección presentamos una breve explicación sobre Datalog. *Nota:* El énfasis sobre Datalog es por su poder descriptivo y no por su poder computacional (como de hecho también fue el caso con el modelo relacional original [5.1]). El objetivo es definir un lenguaje que a fin de cuentas tendrá un poder expresivo mayor que el de los lenguajes relacionales convencionales [23.9]. Por consecuencia, el énfasis en Datalog, y además el énfasis en los sistemas basados en la lógica en general, es muy fuerte sobre las consultas y no sobre las actualizaciones, aunque es posible y necesario extender el lenguaje para que también soporte la actualización (vea más adelante).

En su forma más simple, Datalog soporta la formulación de reglas como las cláusulas de Horn simples sin funciones. En la sección 23.4 definimos una cláusula de Horn para que fuera una WFF de alguna de las dos formas siguientes:

```
A1 AND A2 AND ... AND An

A1 AND A2 AND . . . AND An =*■ B
```

---

*De hecho, hemos definido claramente un *cierre transitivo;* en cualquier momento dado, la relación correspondiente a COMPONENTE_DE es el cierre transitivo de la relación correspondiente a ESTRUCTURA_PARTES (vea el capítulo 6).

(donde las As y *Bs* son ejemplares de predicado no negados que involucran solamente constantes y variables). Sin embargo, siguiendo el estilo de Prolog, Datalog en realidad escribe la segunda de estas formas de la otra manera:

```
B ■*= A1 AND A2 AND        AND An
```

Por lo tanto, para ser consistentes con otras publicaciones en esta área, haremos lo mismo en las partes que vienen a continuación.

En una cláusula como ésta, *B* es la **cabeza de la regla** (o conclusión) y las As son el **cuerpo de la regla** (o premisas o **meta;** cada A individual es una **submeta).** Por brevedad, los ANDs son frecuentemente reemplazados por comas. Un **programa Datalog** es un conjunto de estas cláusulas separadas de alguna forma convencional; por ejemplo, por punto y coma (sin embargo, en este libro no usaremos punto y coma, sino que simplemente comenzaremos cada cláusula en una nueva línea). Dentro de estos programas, no asignamos ningún significado al orden de las cláusulas.

Observe que *toda la "base de datos deductiva "* puede ser considerada como un programa Datalog en el sentido anterior. Por ejemplo, podríamos tomar todos los axiomas establecidos anteriormente para los proveedores y partes (los axiomas base, las restricciones de integridad y los axiomas deductivos), escribirlos en un estilo Datalog, separarlos con punto y coma (o escribirlos en líneas separadas) y el resultado sería un programa Datalog. Sin embargo, como dijimos anteriormente, por lo general *no* especificaremos de esta forma la parte extensional de la base de datos, sino de una forma más convencional. Por lo tanto, el propósito principal de Datalog es soportar específicamente la formulación de axiomas deductivos. Como ya mencionamos, esa función puede ser considerada como una extensión al mecanismo de definición de vistas que se encuentra en los DBMSs relaciónales actuales.

Datalog también puede ser usado como lenguaje de consulta (de nuevo, en forma similar a Prolog). Por ejemplo, supongamos que hemos dado la siguiente definición Datalog de BUEN_PROVEEDOR:

```
BUENPROVEEDOR ( v, vt, ve      <= V ( v, vn, vt, ve )
                                  AND vt > 15
```

Éstas son algunas consultas típicas contra BUEN_PROVEEDOR:

1. Obtener todos los buenos proveedores:

```
? *= BUENPROVEEDOR ( v, vt, ve )
```

2. Obtener los buenos proveedores que están en París:

```
? ■*= BUENPROVEEDOR ( v, vt, París )
```

3. ¿Es un buen proveedor el proveedor VI?

```
? ■*= BUEN_PROVEEDOR ( V1 , vt, ve )
```

Y así sucesivamente. En otras palabras, una consulta en Datalog consiste en una regla especial con una cabeza "?" y un cuerpo que consiste de un solo término que indica el resultado de la consulta; la cabeza "?" significa (por convención) "Desplegar".

"Referencias y bibliografía"). En la presente sección tratamos brevemente algunas de las técnicas más simples y mostramos su aplicación a la consulta "explotar la parte Pl" sobre los siguientes datos de ejemplo:

EP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

Unificación y resolución

Por supuesto, un enfoque posible es usar las técnicas estándar de Prolog para **unificación y resolución,** como las describimos en la sección 23.4. En el ejemplo, este enfoque funciona de la siguiente forma. Las primeras premisas son los axiomas deductivos, que lucen de esta forma (en forma normal conjuntiva):

1. NOT EP (*px, py*) OR COMP (*px, py*)
2. NOT EP (*px, pz*) OR NOT COMP (*pz, py*) OR COMP (*px, py*)

Construimos otra premisa a partir de la conclusión deseada:

3. NOT COMP (P1, *py*) OR RESULTADO (*py*)

Los axiomas base forman las premisas restantes. Considere por ejemplo el axioma base

4. EP (P1, P2)

Si sustituimos Pl en lugar de *px* y P2 en lugar de *py* en la línea 1, podemos resolver las líneas 1 y 4 para que produzcan

5. COMP (P1, P2)

Ahora, si sustituimos P2 en lugar *depy* en la línea 3 y resolviendo las líneas 3 y 5, obtenemos

6. RESULTADO (P2)

Por lo tanto, P2 es un componente de Pl. Un argumento exactamente equivalente mostrará que P3 también es un componente de Pl. Ahora bien, tenemos por supuesto los axiomas adicionales COMP(P1,P2) y COMP(P1,P3); entonces podemos aplicar el proceso anterior en forma recursiva para determinar la explosión completa. Le dejamos los detalles como ejercicio.

Sin embargo, en la práctica, la unificación y resolución pueden ser bastante costosas en rendimiento. Por lo tanto, a menudo será necesario encontrar alguna estrategia más eficiente. Las subsecciones siguientes tratan algunos enfoques posibles para este problema.

Evaluación ingenua

La **evaluación ingenua** [23.25] es probablemente el enfoque más simple de todos. Como su nombre lo sugiere, el algoritmo es muy simple; lo podemos explicar fácilmente (para nuestra consulta de muestra) en términos del siguiente seudocódigo:

```
COMP := EP ;
ejecuta hasta que COMP llegue a un "punto fijo" ;
  COMP := COMP UNION ( COMP » EP ) ; fin ;
DESPLEGAR := COMP WHERE PX = P# ( ' P1 ' ) ;
```

Las varrels COMP y DESPLEGAR (al igual que la varrel EP) tienen cada una dos atributos, PX y PY. A grandes rasgos, el algoritmo funciona formando repetidamente un resultado intermedio que consiste en la unión de la junta de EP con el resultado intermedio anterior, hasta que ese resultado intermedio llega a un **punto fijo;** es decir, hasta que deja de crecer. *Nota:* La expresión "COMP * EP" es una abreviatura para "juntar COMP y EP sobre COMP.PY y EP.PX, y proyectar el resultado sobre COMP.PX y EP.PY"; por brevedad, ignoraremos las operaciones para renombrar atributos que nuestro dialecto de álgebra requeriría para que esta operación funcionara (vea el capítulo 6).

Veamos paso a paso el algoritmo con nuestros datos de ejemplo. Después de la primera iteración del ciclo, el valor de la expresión COMP » EP es como se muestra abajo del lado izquierdo y el valor resultante de COMP es como se muestra abajo del lado derecho (con las tupias que se añaden en esta iteración marcadas con un asterisco):

COMP » EP

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

Después de la segunda iteración se ven de esta forma:

COMP a EP

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6
P1	P6
P2	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6
P1	P6
P2	P6

Observe cuidadosamente que el cálculo de COMP » EP en este segundo paso repite completamente el cálculo de COMP « EP del primer paso y además, calcula algunas tupias adicionales (de hecho, dos tupias adicionales —(Pl ,P6) y (P2,P6)— en este caso). Ésta es una razón por la cual el algoritmo de evaluación ingenua no es muy inteligente.

Después de la tercera iteración, el valor de COMP » EP (después de más cálculos repetidos) resulta ser el mismo que el de la iteración anterior; por lo tanto, COMP ha llegado a un punto fijo y salimos del ciclo. Después el resultado final es calculado como una restricción de COMP:

COMP

PX	PY
P1	P2
P1	P3
P1	P4
P1	P5
P1	P6

Ahora es evidente otra ineficiencia mayúscula: el algoritmo ha calculado efectivamente la explosión para *cada* parte —de hecho, ha calculado el cierre transitivo completo de la relación EP— y luego ha descartado todo nuevamente, con excepción de las tupias que en realidad deseaba; en otras palabras, de nuevo se ha realizado mucho trabajo innecesario.

Cerramos esta subsección señalando que la técnica de evaluación ingenua puede ser considerada como una aplicación del encadenamiento hacia adelante. Comenzando desde la base de datos extensional (es decir, los valores de datos reales), aplica las premisas de la definición (es decir, el cuerpo de la regla) en forma repetida hasta obtener el resultado deseado. De hecho, el algoritmo en realidad calcula el *modelo mínimo* para el programa Datalog (vea las secciones 23.5 y 23.6).

Evaluación semiingenua

La primera mejora obvia al algoritmo de evaluación ingenua es evitar la repetición de los cálculos de cada paso en el siguiente: evaluación semiingenua [23.28]. En otras palabras, en cada paso calculamos solamente las nuevas tupias que necesitan ser añadidas en esta iteración particular. De nuevo, explicamos la idea en términos del ejemplo "explotar la parte Pl". El seudocódigo es:

```
NUEVO  := EP ;
COMP   := NUEVO ;
ejecutar hasta que NUEVO esté vacío ;
         NUEVO  := ( NUEVO » EP) MINUS COMP ;
         COMP   := COMP UNION NUEVO ; fin
; DESPLEGAR : = COMP WHERE PX = P# ( ' P1 ' )
;
```

De nuevo, veamos paso a paso el algoritmo. En la primera entrada al ciclo, NUEVO y COMP son idénticos a EP:

NUEVO

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

Al terminar la primera iteración se ven de esta manera:

NUEVO

PX	PY
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

En esta etapa, COMP es igual a como era en la evaluación ingenua y NUEVO es simplemente las nuevas tupias que fueron añadidas a COMP en esta iteración; observe en particular que NUEVO *no* incluye la tupia (P1 ,P3). Compárelo con la contraparte de la evaluación ingenua.

Al final de la siguiente iteración tenemos:

NUEVO

PX	PY
P1	[illegible]
P2	[illegible]

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6
P1	P6
P2	P6

La siguiente iteración deja vacío a NUEVO y por lo tanto, salimos del ciclo.

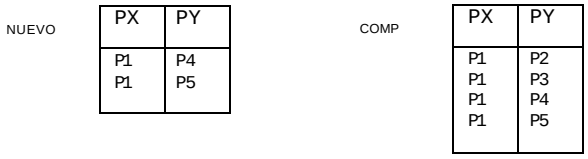
Filtrado estático

El nitrado estático es un refinamiento a la idea básica de la teoría de optimización clásica de realizar restricciones tan pronto como sea posible. Puede ser considerado como una aplicación del encadenamiento hacia atrás en el cual, en efecto, usa información de la consulta (la conclusión) para modificar las reglas (las premisas). También se le conoce como *reducción del conjunto de hechos importantes,* ya que (de nuevo) usa información de la consulta para eliminar inmediatamente tupias inútiles en la base de datos extensional en el conjunto de salida [23.29]. En términos de nuestro ejemplo, podemos explicar el efecto en términos del siguiente seudocódigo:

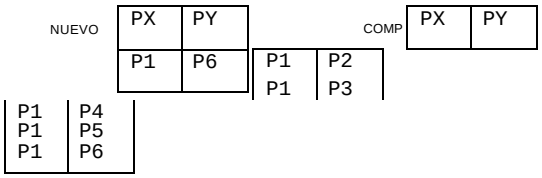
```
NUEVO := EP WHERE PX = P# ( ' P1 ' ) ;
COMP  := NUEVO ;
ejecutar hasta que NUEVO esté vacío ;
         NUEVO  := ( NUEVO * EP ) MINUS COMP ;
         COMP  := COMP UNION NUEVO ;
fin ; DESPLEGAR :- COMP ;
```

De nuevo, veamos paso a paso el algoritmo. En la primera entrada al ciclo, NUEVO y COMP se ven de esta manera:

Al final de la primera iteración se ven de esta manera:



Al final de la siguiente iteración tenemos:



La siguiente iteración deja vacío a NUEVO y por lo tanto, salimos del ciclo.

Esto concluye nuestra breve introducción a las estrategias de procesamiento de consultas recursivas. Por supuesto, en la literatura se han propuesto muchos otros enfoques, la mayoría de ellos mucho más sofisticados que los simples que tratamos anteriormente; sin embargo, en un libro de esta naturaleza es insuficiente el espacio para tratar todo el material de fondo necesario para tener una comprensión adecuada de estos enfoques. Para mayores explicaciones vea por ejemplo las referencias [23.16] a [23.43].

23.8 RESUMEN

Esto nos lleva al final de nuestra corta introducción al tema de las bases de datos que están basadas en la lógica. Aunque las ideas todavía están limitadas en su mayoría al mundo de la investigación, unas cuantas de ellas han comenzado a encontrar su camino en productos relacionales comerciales (este comentario es especialmente cierto con respecto a algunas de las técnicas de optimización). En términos generales, el concepto de bases de datos basadas en la lógica luce interesante; hemos identificado varias ventajas potenciales en diversos puntos de las secciones anteriores. Una ventaja adicional, que no mencionamos explícitamente en el cuerpo del capítulo, es que la lógica podría formar la base de una integración genuinamente suave entre los lenguajes de programación de propósito general y la base de datos. En otras palabras, en vez del enfoque de "sublenguaje de datos incrustado" soportado por los productos SQL actuales

—un enfoque que no es muy elegante, por no decir más— el sistema podría proporcionar un solo lenguaje basado en la lógica y en el cual los "datos son datos", sin tomar en cuenta si están guardados en una base de datos compartida o si son locales a la aplicación. (Por supuesto, hay varios obstáculos que salvar antes de poder lograr este objetivo, y uno no tan pequeño es —en primer lugar— demostrara satisfacción de la comunidad de TI en conjunto, que la lógica es una base adecuada para un lenguaje de programación.)

Revisemos rápidamente los puntos principales del material que hemos cubierto. Comenzamos con un breve tutorial sobre el **cálculo proposicional y** de **predicados,** presentando los siguientes conceptos, entre otros:

- Una **interpretación** de un conjunto de WFFs es la combinación de (a) un universo de dis curso, (b) una transformación entre constantes individuales que aparecen en esas WFFs con objetos de ese universo, y (c) un conjunto de significados definidos para los predicados y funciones que aparecen en esas WFFs.
- Un **modelo** para un conjunto de WFFs es una interpretación en la cual todas las WFFs del conjunto dan como resultado *verdadero.* Un conjunto dado de WFFs puede tener cualquier cantidad de modelos (en general).
- Una **demostración** es el proceso de mostrar que alguna WFF dada *g* (la **conclusión)** es una consecuencia lógica de algún conjunto dado de WFFs/7,/2, ...,*fn* (las **premisas).** Expli camos un método de demostración conocido como **resolución** y unificación con cierto detalle.

Luego examinamos la perspectiva de bases de datos por la **teoría de demostraciones.** En esta perspectiva, la base de datos es considerada como si consistiera en la combinación de una base de datos **extensional** y una base de datos **intensional.** La base de datos extensional contiene **axiomas base,** es decir, los datos base (en general); la base de datos intensional contiene restricciones de integridad y **axiomas deductivos,** es decir, vistas (de nuevo, en general). El "significado" de la base de datos consiste entonces en un conjunto de **teoremas** que pueden ser deducidos a partir de los axiomas; la ejecución de una consulta llega a ser (al menos conceptualmente) un proceso de **demostración de teoremas. Un DBMS deductivo** es un DBMS que soporta esta perspectiva de la teoría de demostraciones. Describimos brevemente a **Datalog,** un lenguaje de usuario para estos DBMSs.

Una diferencia inmediata entre Datalog y los lenguajes relaciónales tradicionales es que Datalog soporta axiomas **recursivos** y por lo tanto consultas recursivas, aunque no hay razón por la cual el álgebra y el cálculo relaciónales tradicionales no deban ser extendidos para que hagan lo mismo (vea la explicación del operador TCLOSE en el capítulo 6).* Tratamos algunas técnicas simples para la evaluación de dichas consultas.

En conclusión: iniciamos este capítulo mencionando varios términos —"bases de datos lógicas", "DBMS inferencial", "DBMS deductivo", etcétera— que a menudo se encuentran en la literatura de investigación (y hasta cierto punto, incluso en los anuncios de los fabricantes). Por

*Con respecto a esto, es interesante observar que los DBMSs relaciónales deben tener de todas formas la posibilidad de realizar procesamiento recursivo en segundo plano, debido a que el catálogo contendrá cierta información estructurada en forma recursiva (las definiciones de vistas expresadas en términos de otras definiciones de vistas son uno de estos casos).

lo tanto, cerremos el capítulo proporcionando algunas definiciones para esos términos. Sin embargo, queremos prevenirlo que no siempre hay consenso sobre estos temas, por lo que es probable que encuentre diferentes definiciones en la literatura. Las siguientes definiciones son mis preferidas:

- *Procesamiento de consultas recursivas.* Ésta es fácil. El procesamiento de consultas recursivas se refiere a la evaluación y, en particular, a la optimización de consultas cuya definición es intrínsecamente recursiva (vea la sección 23.7).
- *Base de conocimientos.* Este término a menudo es usado con el significado de lo que en la sección 23.6 llamamos la base de datos intensional; es decir, consiste en las *reglas* (las restricciones de integridad y axiomas deductivos), en oposición a los datos básicos los cuales constituyen la base de datos extensional. Pero otros escritores usan ocasionalmente "base de conocimientos" para que signifique la combinación de las bases de datos intensional y extensional (vea a continuación "base de datos deductiva"); con excepción de que, como lo dice la referencia [23.10], "una base de conocimientos con frecuencia incluye objetos complejos [así como] relaciones clásicas" (vea la parte VI de este libro para una explicación sobre los "objetos complejos"). Luego, el término tiene de nuevo otro significado más específico en los sistemas de lenguaje natural. Probablemente lo mejor es evitarlo por completo.
- *Conocimientos.* ¡Otra fácil! Conocimientos es lo que hay en la base de conocimientos... Por lo tanto, esta definición reduce el problema de la definición de "conocimientos" a un problema anterior no resuelto.
- *KBMS (Sistemas de Administración de Base de Conocimientos).* Es el software que administra la base de conocimientos. Por lo general, el término se usa como un sinónimo para el DBMS deductivo (vea el siguiente párrafo).
- *DBMS deductivo.* Es un DBMS que soporta la perspectiva de bases de datos por la teoría de demostraciones y que en particular es capaz de deducir información adicional a partir de la base de datos extensional mediante la aplicación de reglas de inferencia (es decir, deductivas) que están guardadas en la base de datos intensional. Un DBMS deductivo casi seguramente soportará reglas recursivas y por lo tanto, realizará procesamiento de consultas recursivas.
- *Base de datos deductiva* (término desaprobado). Es una base de datos que es administrada por un DBMS deductivo.
- *DBMS experto.* Es un sinónimo para el DBMS deductivo.
- *Base de datos experta* (término desaprobado). Es una base de datos que es administrada por un DBMS experto.
- *DBMS inferencial.* Es un sinónimo para el DBMS deductivo.
- *Sistema basado en la lógica.* Es un sinónimo para el DBMS deductivo.
- *Base de datos lógica* (término desaprobado). Es un sinónimo para la base de datos deductiva.
- *La lógica como un modelo de datos.* Es un modelo de datos que consiste (al menos) en objetos, reglas de integridad y operadores. En un DBMS deductivo, los objetos, las reglas de integridad y los operadores están representados de la misma manera uniforme, concretamente como axiomas en un lenguaje lógico como Datalog; de hecho, como explicamos en la sección 23.6, una base de datos en un sistema de éstos puede ser considerada precisamente como un programa lógico que contiene axiomas de los tres tipos. Por lo tanto, en un sistema de éstos podemos decir legítimamente que el modelo de datos abstracto para el sistema *es* la lógica misma.

EJERCICIOS

23.1 Use el método de resolución para ver si las siguientes metadeclaraciones constituyen demostra ciones válidas en el cálculo proposicional:

a. *A* ⇒ S, *C* ⇒ *B, D* =*■ (A* OR *C*) , *D* \- *B*

b. *(A* ⇒ *B)* AND (C ⇒ *D) , (B* => *E* AND *D* =*■ F),*
NOT (*E* AND F) , *A* ⇒ C I- NOT *A*

C. (,4 OR 8) ⇒• 0, D ⇒• NOT (*E* OR *F*) , NOT (*B* AND C AND £)
\- NOT (G ⇒• NOT (C AND *H))*

23.2 Convierta las siguientes WFFs a forma clausal:

a. FORALL x (FORALL y
(p (*X, y*) ⇒ EXISTS z (</ (x, *z*))))

b. EXISTS x (EXISTS y
(p (x, y) ⇒ FORALL z (*q* (x, z))))

C. EXISTS x (EXISTS y
(*p* (x, y) ⇒ EXISTS *z (q (x, z))))*

23.3 El siguiente es un ejemplo estándar de una base de datos lógica:

```
HOMBRE ( Adán )
MUJER ( Eva )
HOMBRE ( Caín )
HOMBRE ( Abel )
HOMBRE ( Enoc )

PROGENITOR ( Adán, Caín )
PROGENITOR ( Adán, Abel )
PROGENITOR ( Eva, Caín )
PROGENITOR ( Eva, Abel )
PROGENITOR ( Caín, Enoc )

PADRE ( X, y ) ■*= PROGENITOR ( x, y ) AND HOMBRE ( x )
MADRE ( x, y ) ■*= PROGENITOR ( x, y ) AND MUJER ( x )

FRATERNO ( X, y ) *- PROGENITOR ( z, X ) AND PROGENITOR ( z, y )

HERMANO ( x, y ) •*= FRATERNO ( X, y ) AND HOMBRE ( X ) HERMANA (

X, y ) ■*= FRATERNO ( x, y ) AND MUJER ( X )

ANCESTRO ( x, y ) ■«= PROGENITOR ( x, y )

ANCESTRO ( x, y ) ■*= PROGENITOR ( X, z ) AND ANCESTRO { z, y )
```

Use el método de resolución para responder las siguientes consultas:

a. ¿Quién es la madre de Caín?
b. ¿Quiénes son los fraternos de Caín?
c. ¿Quiénes son los hermanos de Caín?
d. ¿Quiénes son las hermanas de Caín?
e. ¿Quiénes son los ancestros de Enoc?

23.4 Defina los términos *interpretación* y *modelo.*

23.5 Escriba un conjunto de axiomas Datalog para la parte de definición (solamente) de la base de datos de proveedores, partes y proyectos.

23.6 Dé soluciones Datalog, donde sea posible, para los ejercicios 6.13 a 6.50.

23.7 Dé soluciones Datalog, donde sea posible, para el ejercicio 8.1.

23.8 Complete, para su propia satisfacción, la explicación que dimos en la sección 23.7 sobre laimplementación de la unificación y resolución de la consulta "Explotar la parte Pl".

REFERENCIAS Y BIBLIOGRAFÍA

El campo de los sistemas basados en la lógica ha crecido inmensamente a lo largo de los últimos años; la siguiente lista representa una pequeña fracción de la literatura disponible actualmente. Está acomodada parcialmente en grupos, de la siguiente forma:

- Las referencias [23.1] a [23.9] son libros que están dedicados al tema de la lógica en general (en particular en un contexto de computación o de base de datos) o son conjuntos de artículos sobre bases de datos basadas específicamente en la lógica.
- Las referencias [23.10] a [23.12] son tutoriales, así como los libros de Ceri *et al.* [23.46] y Das [23.47].
- Las referencias [23.14], [23.17] a [23.20], [23.30], [23.49] y [23.50] se refieren a la operación de cierre transitivo y su implementación.
- Las referencias [23.21] a [23.24] describen una técnica importante de procesamiento de consultas recursivas llamada *conjuntos mágicos* (y sus variaciones). *Nota:* Al respecto vea también las refe rencias [17.24] a [17.26].

Las referencias restantes se incluyen principalmente para mostrar qué tanta investigación se está realizando en este campo; tratan una variedad de aspectos sobre el tema y son presentadas, en su mayoría, sin comentarios adicionales.

23.1 Robert R. Stoll: *Sets, Logic, and Axiomatic Theories.* San Francisco, Calif.: W. H. Freeman and Company (1961).

Una buena introducción a la lógica en general.

23.2 Zohar Manna y Richard Waldinger: *The Logical Basis for Computer Programming—Volume 1: Deductive Reasoning* (1985); *Volume II: Deductive Techniques* (1990). Reading, Mass.: Addison-Wesley (1985, 1990).

23.3 Peter M. D. Gray: *Logic, Algebra and Databases.* Chichester, England: Ellis Horwood Ltd. (1984).

Contiene una introducción accesible sobre el cálculo proposicional y sobre el cálculo de predicados (entre varios otros temas relevantes) desde un punto de vista de base de datos.

23.4 Adrian Walker, Michael McCord, John F. Sowa y Walter G. Wilson: *Knowledge Systems and Prolog* (2a. edición). Reading, Mass.: Addison-Wesley (1990).

Este libro es acerca de la programación lógica en general y no específicamente sobre base de datos basadas en la lógica, pero contiene mucho material que es importante para este último tema.

23.5 Hervé Gallaire y Jack Minker: *Logic and Data Bases.* Nueva York, N.Y.: Plenum Publishing Corp. (1978).

Una de las primeras, si no es que *la* primera, colecciones sobre artículos de este tema.

23.6 Larry Kershberg (ed.): *Expert Database Systems* (Proc. 1st Int. Workshop on Expert Database Systems, Kiawah Island, S.C.). Menlo Park, Calif.: Benjamin/Cummings (1986).

Es una excelente e incitadora colección de artículos basados en ideas. Sin embargo, no todos están relacionados directamente con el tema principal de este capítulo. Además, ¡los títulos de las secciones tienen un cierto grado de confusión sobre lo que en realidad es el tema de "sistemas expertos de bases de datos". Los títulos son los siguientes:

1. Teoría de las bases de conocimientos.
2. La programación lógica y las bases de datos.
3. Arquitecturas, herramientas y técnicas de sistemas de bases de datos expertos.
4. El razonamiento en los sistemas de bases de datos expertos.
5. Acceso e interacción con bases de datos inteligentes.

Además, hay un artículo introductorio de John Smith sobre los sistemas de base de datos expertos e informes de grupos de trabajo sobre (1) sistemas de administración de bases de conocimientos, (2) la programación lógica y las bases de datos y (3) los sistemas de bases de datos de objetos y los sistemas de conocimientos. Como indica Kershberg en su prefacio, el concepto de sistema de base de datos experto "connota diversas definiciones y decididamente diferentes arquitecturas".

23.7 Jack Minker (ed.): *Foundations of Deductive Databases and Logic Programming.* San Mateo, Calif.: Morgan Kaufmann (1988).

23.8 John Mylopoulos y Michael L. Brodie (eds.): *Readings in Artificial Intelligence and Databases.* San Mateo, Calif.: Morgan Kaufmann (1988).

23.9 Jeffrey D. Ullman: *Database and Knowledge-Base Systems* (en dos volúmenes). Rockville, Md.: Computer Science Press (1988, 1989).

El volumen I de esta obra de dos volúmenes incluye un (largo) capítulo (de un total de 10) que está completamente dedicado al enfoque basado en la lógica. Ese capítulo (que dicho sea de paso, es el origen de Datalog) incluye una explicación sobre el vínculo entre la lógica y el álgebra relacional, y otro sobre cálculo relacional —en las versiones de dominio y tupia— como un caso especial del enfoque lógico. El volumen II incluye cinco capítulos (de siete) sobre diversos aspectos de las bases de datos basadas en la lógica.

23.10 Georges Gardarin y Patrick Valduriez: *Relational Databases and Knowledge Bases.* Reading, Mass.: Addison-Wesley (1989).

Contiene un capítulo sobre sistemas deductivos que (aunque es de naturaleza tutorial) presenta la teoría subyacente, algoritmos de optimización, etcétera, con mucho mayor detalle que el que damos en el presente capítulo.

23.11 Michael Stonebraker: *Introduction to* "Integration of Knowledge and Data Management", en Michael Stonebraker (ed.), *Readings in Database Systems.* San Mateo, Calif.: Morgan Kaufmann (1988).

23.12 Hervé Gallaire, Jack Minker y Jean-Marie Nicolas: "Logic and Databases: A Deductive Approach", *ACM Comp. Surv. 16,* No. 2 (junio, 1984).

23.13 Veronica Dahl: "On Database Systems Development through Logic", *ACM TODS 7,* No. 1 (marzo, 1982).

Es una descripción buena y clara de las ideas básicas subyacentes en las bases de datos basadas en la lógica, con ejemplos tomados de un prototipo basado en Prolog que fue implementado por Dahl en 1977.

23.14 Rakesh Agrawal: "Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries", *IEEE Transactions on Software Engineering 14,* No. 7 (julio, 1988).

Propone un nuevo operador llamado *alpha* que soporta la formulación de "una gran clase de consultas recursivas" (de hecho, un superconjunto de consultas recursivas lineales), mientras permanece dentro del marco de trabajo del álgebra relacional convencional. La aseveración es que el operador *alpha* es suficientemente poderoso para manejar la mayoría de los problemas prácticos que involucran recursion, y al mismo tiempo es más fácil de implementar de manera eficiente de lo que sería un mecanismo de recursion completamente general. El artículo da varios ejemplos sobre el uso del operador propuesto; en particular, muestra la manera en que pueden ser fácilmente manejados los problemas de cierre transitivo y de "requerimientos brutos" (vea la referencia [23.17] y la sección 23.6, respectivamente).

La referencia [23.19] describe un trabajo relacionado con la implementación. La referencia [23.18] también es importante.

23.15 Raymond Reiter: "Towards a Logical Reconstruction of Relational Database Theory", en Michael L. Brodie, John Mylopoulos y Joachim W. Schmidt (eds.), *On Conceptual Modelling: Pers pectives from Artificial Intelligence, Databases, and Programming Languages.* Nueva York, N.Y.: Springer Verlag (1984).

Como mencionamos en la sección 23.2, el trabajo de Reiter no fue el primero en esta área—muchos investigadores habían averiguado el vínculo entre la lógica y las bases de datos (vea por ejemplo las referencias [23.5], [23.7] y [23.13])— pero parece que ha sido la "reconstrucción lógica de la teoría relacional" de Reiter la que provocó gran parte de la actividad subsecuente y el alto grado de interés actual sobre el campo.

23.16 Francois Bancilhon y Raghu Ramakrishnan: "An Amateur's Introduction to Recursive Query Processing Strategies", Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (mayo, 1986). Vuelto a publicar en forma revisada en Michael Stonebraker (ed.), *Readings in Database Systems.* San Mateo, Calif.: Morgan Kaufmann (1988). También vuelto a publicar en la re ferencia [23.8].

Es una panorámica excelente. El artículo comienza observando que hay lados positivos y negativos en toda la investigación del problema de implementación de consultas recursivas. El lado positivo es que han sido identificadas muchas técnicas que al menos resuelven el problema; el lado negativo es que no queda claro del todo cómo seleccionar la técnica más adecuada en una situación dada (en particular, la mayoría de las técnicas se presentan en la literatura con muy poca o ninguna explicación de las características de rendimiento). Luego, después de una sección que describe las ideas básicas de las bases de datos lógicas, el artículo describe varios algoritmos propuestos, tales como la evaluación ingenua, evaluación semiingenua, consulta/subconsulta iterativa, consulta/subconsulta recursiva, APEX, Prolog, Henschen/Naqvi, Aho-Ullman, Kifer-Lozinskii, conteo, conjuntos mágicos y conjuntos mágicos generalizados. El artículo compara estos enfoques diferentes a partir del dominio de aplicación (es decir, la clase de problemas a los cuales es posible aplicar el algoritmo en forma útil), el rendimiento y la facilidad de implementación. El artículo también incluye cifras de rendimiento (con análisis comparativos) para probar los diversos algoritmos con una prueba simple.

23.17 Yannis E. Ioannidis: "On the Computation of the Transitive Closure of Relational Operators", Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japón (agosto, 1986).

El cierre transitivo es una operación de importancia fundamental en el procesamiento de consultas recursivas [23.18]. Este artículo propone un nuevo algoritmo (basado en el enfoque de "divide y vencerás") para la implementación de esa operación. Vea también las referencias [23.14], [23.18] a [23.20], [23.49] y [23.50].

23.18 H. V. Jagadish, Rakesh Agrawal y Linda Ness: "A Study of Transitive Closure as a Recursion Mechanism", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

Para citar el resumen: "[Este artículo muestra] que toda consulta linealmente recursiva puede ser expresada como un cierre transitivo precedido y seguido (posiblemente) por operaciones que ya están disponibles en el álgebra relacional." Por lo tanto, los autores sugieren que proporcionar una implementación eficiente del cierre transitivo es suficiente como base para proporcionar una implementación eficiente de la recursion lineal en general y por lo tanto, para hacer DBMSs deductivos eficientes en una gran clase de problemas recursivos.

23.19 Rakesh Agrawal y H. Jagadish: "Direct Algorithms for Computing the Transitive Closure of Database Relations", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (sep tiembre, 1987).

Propone un conjunto de algoritmos de cierre transitivo que "no ven el problema como el de evaluar una recursion, sino que en su lugar obtienen el cierre a partir de los primeros principios" (y de ahí el término *directo).* El artículo incluye un resumen útil de trabajos anteriores sobre otros algoritmos directos.

23.20 Hongjun Lu: "New Strategies for Computing the Transitive Closure of a Database Relation", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

Más algoritmos para el cierre transitivo. Al igual que la referencia [23.19], el artículo también incluye un estudio útil sobre enfoques anteriores para el problema.

23.21 Francois Bancilhon, David Maier, Yehoshua Sagiv y Jeffrey D. Ullman: "Magic Sets and Other Strange Ways to Implement Logic Programs", Proc. 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1986).

La idea básica de los "conjuntos mágicos" es presentar de manera dinámica nuevos conjuntos de reglas ("reglas mágicas") que garantizan producir el mismo resultado que la consulta original pero son más eficientes en el sentido de que reducen el conjunto de "hechos importantes" (vea la sección 23.7). Los detalles son un poco complejos y están fuera del alcance de estas notas; consulte el artículo o el estudio de Bancilhon y Ramakrishnan [23.16] o los libros de Ullman [23.9] o Gardarin y Valduriez [23.10] donde encontrará explicaciones más amplias. Hacemos notar que subsecuentemente se han visto numerosas variaciones a la idea básica; vea por ejemplo, las referencias [23.22] a [23. 24] a continuación. Vea también las referencias [17.24] a [17.26].

23.22 Catriel Beeri y Raghu Ramakrishnan: "On the Power of Magic", Proc. 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems (1987).

23.23 Domenico Sacca y Carlo Zaniolo: "Magic Counting Methods", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

23.24 Georges Gardarin: "Magic Functions: A Technique to Optimize Extended Datalog Recursive Pro grams", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

23.25 A. Aho y J. D. Ullman: "Universality of Data Retrieval Languages", Proc. 6th ACM Sympo sium on Principles of Programming Languages, San Antonio, Tx. (enero, 1979).

Dada una secuencia de relaciones *R,f(R),f(fiR)),* ...(donde/es alguna función fija), el **menor punto fijo** de la secuencia está definido para que sea una relación *R** derivada de acuerdo con el siguiente algoritmo de evaluación ingenua (vea la sección 23.7):

```
fl* := R ;
ejecutar hasta que R* deje de crecer ;
         R* := fl* UNION f(S*) ; fin
;
```

Este artículo propone la incorporación de un operador de menor punto fijo para el álgebra relacional.

23.26 Jeffrey D. Ullman: "Implementation of Logical Query Languages for Databases", *ACM TODS 10,* No. 3 (septiembre, 1985).

Describe una clase importante de técnicas de implementación para consultas posiblemente recursivas. Las técnicas están definidas en términos de "reglas de captura" sobre "árboles de regla/meta", los cuales son grafos que representan una estrategia de consulta en términos de cláusulas y predicados. El artículo define varias de estas reglas; una que corresponde a la aplicación de los operadores del álgebra relacional, dos más que corresponden al encadenamiento hacia adelante o hacia atrás, respectivamente, y una regla "lateral" que permite que los resultados sean pasados de una submeta a otra. El paso de información lateral se convierte después en la base para las técnicas llamadas *conjuntos mágicos* [23.21] a [23.24].

23.27 Shalom Tsur y Carlo Zaniolo: "LDL: A Logic-Based Data-Language", Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japón (agosto, 1986).

LDL incluye (1) un generador de tipo "conjunto", (2) una negación (basada en diferencia de conjuntos), (3) operaciones de definición de datos y (4) operaciones de actualización. Es un lenguaje puro (no hay dependencias de ordenamiento entre las declaraciones) y compilado, no interpretado. Vea también el libro de Naqvi y Tsur [23.45] sobre el mismo tema.

23.28 Francois Bancilhon: "Naive Evaluation of Recursively Defined Relations", en Michael Brodie y John Mylopoulos (eds.), *On Knowledge Base Management Systems: Integrating Database andAI Systems.* Nueva York, N.Y.: Springer Verlag (1986).

23.29 Eliezer L. Lozinskii: "A Problem-Oriented Inferential Database System", *ACM TODS 11,* No. 3 (septiembre, 1986).

Es el origen del concepto "hechos importantes". El artículo describe un prototipo de sistema que utiliza la base de datos extensional para limitar la expansión muy rápida del espacio de búsqueda que, por lo general, generan las técnicas de inferencia.

23.30 Arnon Rosenthal *et ai:* "Traversal Recursion: A Practical Approach to Supporting Recursive Applications", Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (junio, 1986).

23.31 Georges Gardarin y Christophe de Maindreville: "Evaluation of Database Recursive Logic Pro grams as Recurrent Function Series", Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (junio, 1986).

23.32 Louiqa Raschid y Stanley Y. W. Su: "A Parallel Processing Strategy for Evaluating Recursive Queries", Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japón (agosto, 1986).

23.33 Nicolas Spyratos: "The Partition Model: A Deductive Database Model", *ACM TODS 12,* No. 1 (marzo, 1987).

23.34 Jiawei Han y Lawrence J. Henschen: "Handling Redundancy in the Processing of Recursive Queries", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

23.35 Weining Zhang y C. T. Yu: "A Necessary Condition for a Doubly Recursive Rule to be Equiva lent to a Linear Recursive Rule", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

23.36 Wolfgang Nejdl: "Recursive Strategies for Answering Recursive Queries—The RQA/FQI Stra tegy", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

23.37 Kyu-Young Whang y Shamkant B. Navathe: "An Extended Disjunctive Normal Form Approach for Optimizing Recursive Logic Queries in Loosely Coupled Environments", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

23.38 Jeffrey F. Naughton: "Compiling Separable Recursions", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. (junio, 1988).

23.39 Cheong Youn, Lawrence J. Henschen y Jiawei Han: "Classification of Recursive Formulas in Deductive Databases", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. (junio, 1988).

23.40 S. Ceri, G. Gottlob y L. Lavazza: "Translation and Optimization of Logic Queries: The Alge braic Approach", Proc. 12th Int. Conf. on Very Large Data Bases, Kyoto, Japón (agosto, 1986).

23.41 S. Ceri y L. Tanca: "Optimization of Systems of Algebraic Equations for Evaluating Datalog Queries", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

23.42 Allen Van Gelder: "A Message Passing Framework for Logical Query Evaluation", Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (junio, 1986).

23.43 Ouri Wolfson y Avi Silberschatz: "Distributed Processing of Logic Programs", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. (junio, 1988).

23.44 Jeffrey F. Naughton *et al.:* "Efficient Evaluation of Right-, Left-, and Multi-Linear Rules", Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (junio, 1989).

23.45 Shamim Naqvi y Shalom Tsur: *A Logical Language for Data and Knowledge Bases.* Nueva York, N.Y.: Computer Science Press (1989).

Es una presentación profunda, del tamaño de un libro, sobre el lenguaje LDL [23.27].

23.46 S. Ceri, G. Gottlob y L. Tanca: *Logic Programming and Databases.* Nueva York, N.Y.: Springer Verlag (1990).

23.47 Subrata Kumar Das: *Deductive Databases and Logic Programming.* Reading, Mass.: Addison-Wesley(1992).

23.48 Michael Kifer y Eliezer Lozinskii: "On Compile Time Query Optimization in Deductive Data bases by Means of Static Filtering", *ACM TODS 15,* No. 3 (septiembre, 1990).

23.49 Rakesh Agrawal, Shaul Dar y H. V. Jagadish: "Direct Transitive Closure Algorithms: Design and Performance Evaluation", *ACM TODS 15,* No. 3 (septiembre, 1990).

23.50 H. V. Jagadish: "A Compression Method to Materialize Transitive Closure", *ACM TODS 15,* No. 4 (diciembre, 1990).

Propone una técnica de indexación que permite que el cierre transitivo de una relación dada sea guardado en forma comprimida, para que la prueba de ver si una tupia dada aparece en el cierre, pueda ser realizada por medio de una sola búsqueda en tabla seguida de una comparación del índice.

23.51 Serge Abiteboul y Stéphane Grumbach: "A Rule-Based Language with Functions and Sets", *ACM TODS 16,* No. 1 (marzo, 1991).

Describe un lenguaje llamado COL ("lenguaje de objetos complejos") —una extensión de Datalog— que integra las ideas de las bases de datos deductivas y de objetos.

RESPUESTAS A EJERCICIOS SELECCIONADOS

23.1 a. Válida, b. Válida, c. No válida

23.2 En lo que viene a continuación, *a* , *b y c* son constantes de Skolem y/es una función de Skolem con dos argumentos.

a. *p (x, y)■ m*. q (x, f.(*, y))*

b. *p (a, b) => q (a, z)*

C. *p* (a, *b*) =► q (*a, c)*

23.6 De acuerdo con lo que hacemos usualmente, hemos numerado las siguientes soluciones como 23.6.n, donde *6.n* es el número del ejercicio original en el capítulo 6.

23.6.13 *7'<*-* **Y** (**y,** y/7, ye)

23.6.14 ? •«= Y (y, yn, Londres)

23.6.15 RES **(*v*) ■« = VPY (*v, p,* Y1)**
? ■«= RES (i/)

23.6.16 ? ■»•* VPY **(*v, p,* y, q)** AND 300 **< <J** AND ***q*** **<** 750

23.6.17 RES **(*pl, pe)* <= P (p, pn, pJ, pp, pe)**
? ■<= RES **(*pl, pe*)**

23.6.18 RES **(*v, p, y*) ■*= *V (v, vn, vt, C)* AND**
P (p, *pn, pl, pp, c)* AND
Y (y, *yn, c)*
? ■<= RES **(*v, p, y*)**

23.6.19 y 23.6.20 No se pueden realizar sin negación.

23.6.21 RES (p) ■*= VPY (*v, p, y, q)* AND
V (*v, vn, vt,* Londres)
? -*t*= RES (p)

23.6.22 RES (p) ■«= VPY (*v, p,* y, q) AND
V (v, vn, vt, Londres) AND
v (y> *y*n> Londres)
? ■«= RES (p)

23.6.23 RES (O, c2) ■«= VPY (*v, p, y, q*) AND
V (*v, vn, vt, d)* AND
Y (y, yn, *c2)*
? •«= RES (*C1, C2)*

23.6.24 RES (*p*) ■«= VPY (*v, p,* y, q) AND
V (v, *vn, vt, c*) AND
V (y, yn, c)
? ■«= RES (p)

23.6.25 No se puede realizar sin negación.

23.6.26 RES **(*p1, p2*) ■«= VPY (*v, pl, y1, q1*)** AND
VPY **{ *v, p2, y2, q2*) ? <=**
RES **(*pl, p2*)**

23.6.27 a 23.6.30 No se pueden realizar sin agrupamiento y agregación.

```
23.6.31 RES ( yn ) •«- Y ( y, yn, ye ) AND
                       VPY ( V1 , p, y, q )
        ? ■*- RES ( yn )

23.6.32 RES ( pl ) ■*= P ( p, pn, pl, pp, pe ) AND
                       VPY ( V1, p, y, q )
        ? ■«= RES ( pl )

23.6.33 RES ( p ) <= P ( p, pn, pl, pp, pe ) AND
                      VPY ( v, p, y, q ) AND
                      Y ( y> Y'i Londres )
        ? ■*= RES ( p )

23.6.34 RES ( y ) «- VPY ( v, p, y, q ) AND
                       VPY ( V1, p, y2, q2 )
        ? ■«= RES ( y )

23.6.35 RES ( v ) -«= VPY ( v, p, y, q ) AND
                       VPY ( v2, p, y2, q2 ) AND
                       VPY ( v2, p2, y3, q3 ) AND
                       P ( p2, pn, Rojo, pp, c )
        ? -«= RES ( v )

23.6.36 RES ( V ) ■«- V ( V, ra, i/t, c ) AND
                       V ( V1, i/ní, vtr, cí ) AND vt < vt1
        T <- RES ( V )
```

23.6.37 a 23.6.39 No se pueden realizar sin agrupamiento y agregación.

23.6.40 a 23.6.44 No se pueden realizar sin negación.

```
23.6.45 RES ( c ) -4= V ( v, vn, vt, c )
        RES ( c ) •«= P ( p, pn, pl, pp, c )
        RES ( c ) -4= Y ( y, yn, c )
        ? ■*= RES ( c )

23.6.46 RES ( p ) ■<= VPY ( v, p, y, q ) AND
                       V ( v, vn, vt, Londres )
        RES ( p ) •*= VPY ( v, p, y, q ) AND
                       Y ( yi y, Londres )
        ? ■«= RES ( p )
```

23.6.47 y 23.6.48 No se pueden realizar sin negación.

23.6.49 y 23.6.50 No se pueden realizar sin agrupamiento.

23.7 Mostramos las restricciones como implicaciones convencionales, en lugar del estilo "hacia atrás" de Datalog.

```
a. CIUDAD ( Londres )
   CIUDAD ( París )
   CIUDAD ( Roma )
   CIUDAD ( Atenas )
   CIUDAD ( Oslo )
   CIUDAD ( Estocolmo )
   CIUDAD ( Madrid )
   CIUDAD ( Amsterdam )

   V ( v, vn, vt, c )      =p- CIUDAD ( c )
   P ( p, pn, pl, pp, c ) =►• CIUDAD ( c )
   Y ( y, yn, c )          =► CIUDAD ( c )
```

b. No puede realizarse sin operadores escalares adecuados.

C. P (*p*, *pn*, Rojo, *pp*, *pe*) =► *pp* < 50

e. d. No puede realizarse sin operadores de negación o de totales.

V (*v1*, *vn1*, *vt1*, Atenas) AND
V (*v2*, *vn2*, *vt2*, Atenas) =► *v1* • *v2*

f. No puede realizarse sin agolpamiento y agregación.

g. No puede realizarse sin agrupamiento y agregación.

h. Y (*y*, *yn*, *c*) =»■ V (*v*, *vn*, *vt*, *c*)

i. Y (y, *yn*, *c*) =► VPY (*v*, *p*, *y*, *q*) AND V (*v*, *vn*, *vt*, *c*)

j. P (*p1*, *pn1*, *pl1*, *pp1*, *pd*) =► P (*p2*, *pn2*, Rojo, *pp2*, *pc2*)

k. No puede realizarse sin operadores de totales.

1. V (*v*, *vn*, *vt*, Londres) *m»* VPY (*v*, P2, y, *q*)

m. P (*p1*, *pn1*, *pit*, *pp1*, *pd*) ■*

P (*p2*, *pn2*, Rojo, *pp2*, *pc2*) AND *pp2* < 50

n. No puede realizarse sin operadores de totales.

o. No puede realizarse sin operadores de totales.

p. No puede realizarse (es una restricción de transición).

q. No puede realizarse (es una restricción de transición).

PARTE VI

BASES DE DATOS DE OBJETOS Y DE OBJETOS/RELACIÓNALES

La tecnología de objetos es una disciplina importante en el campo de la ingeniería de software en general; por lo tanto, es natural preguntarnos si es importante para el campo de la administración de bases de datos en particular, y de ser así, cuál es su relevancia. Sin embargo, ¡no hay consenso en las respuestas a estas preguntas! Algunas autoridades creen que los sistemas de bases de datos de objetos dominarán el mundo y reemplazarán por completo a los sistemas relacionales, pero otros creen que sólo son adecuados para determinados problemas muy específicos y nunca capturarán más que una pequeña fracción del mercado total. Más recientemente han comenzado a aparecer sistemas que soportan una "tercera vía": sistemas que integran las tecnologías de objetos y las relacionales en un intento por obtener lo mejor de ambos mundos. Los dos capítulos de esta parte final del libro examinan dichos temas a profundidad; el capítulo 24 trata los sistemas de objetos puros y el capítulo 25 trata a los sistemas "de objetos/relacionales" más recientes.

CAPITULO 24

Bases de datos de objetos

24.1 INTRODUCCIÓN

En la década pasada se generó mucho interés sobre los sistemas de bases de datos *orientadas a objetos* **(sistemas de objetos,** para abreviar). Los sistemas de objetos son considerados por algunas personas como serios competidores de los sistemas relaciónales (o sistemas SQL, en todo caso), al menos para determinados tipos de aplicaciones. En este capítulo examinamos en detalle los sistemas de objetos, presentamos y explicamos conceptos básicos de objetos, analizamos esos conceptos a profundidad y proporcionamos algunas opiniones con respecto a la conveniencia de incorporar tales conceptos en los sistemas de bases de datos del futuro.

¿Por qué hay actualmente tanto interés en los sistemas de objetos? Bueno, "todos saben" que los sistemas SQL clásicos son inadecuados en diversas formas. Y algunas personas —¡aunque yo no!— argumentarán que la teoría subyacente (es decir, el modelo relacional) también es inadecuada. Sea como fuere, algunas de las nuevas características que parecen ser necesarias en los DBMS han existido durante muchos años en los *lenguajes de programación de objetos* —como C++ y Smalltalk— y por lo tanto, es natural investigar la idea de incorporar esas características en los sistemas de bases de datos. Y muchos investigadores, así como varios fabricantes, han hecho exactamente eso.

Por lo tanto, los sistemas de objetos tienen sus orígenes en los lenguajes de programación de objetos. Y la idea básica es la misma en ambos casos; es decir: los usuarios no tienen que luchar contra las construcciones orientadas a la máquina —como los bits y los bytes (e incluso los campos y registros)—, sino que en su lugar deben tener la posibilidad de manejar **objetos** y **operaciones** sobre esos objetos, que se asemejan mucho más a sus contrapartes en la realidad. Por ejemplo, en lugar de tener que pensar en términos de una "tupia DEPTO" y además en una colección de "tupias EMP" correspondientes que incluyan "valores de clave externa" que "hagan referencia" al "valor de clave primaria" de esa "tupia DEPTO", el usuario debe pensar directamente en términos de un *objeto departamento* que en realidad contenga un conjunto correspondiente de *objetos empleado*. Y en lugar de, por ejemplo, tener que "insertar" una "tupia" en la "varrel EMP" con un "valor de clave externa" adecuado que "haga referencia" al "valor de clave primaria" de alguna "tupia" en la "varrel DEPTO", el usuario debe *contratar* un objeto empleado directamente en el objeto departamento relevante. En otras palabras, la idea fundamental es **elevar el nivel de abstracción.**

Ahora bien, elevar el nivel de abstracción es incuestionablemente un objetivo valioso y el paradigma de objetos ha tenido mucho éxito para satisfacer ese objetivo en los lenguajes de programación [24.15]. Por lo tanto, tiene sentido preguntarnos si el mismo paradigma también puede ser aplicado satisfactoriamente en las bases de datos. Además, la idea de manejar una base de datos que está compuesta de "objetos complejos" (por ejemplo, los objetos departamento que "saben lo que significa" contratar a un empleado, cambiar a su gerente o reducir su presupuesto) —en lugar

de tener que manejar "varrels" e "inserción de tupias" y "claves externas", etcétera— es obviamente más atractiva desde el punto de vista del usuario; al menos a primera vista.

Sin embargo, aquí son adecuadas unas palabras de precaución. El punto es que aunque las disciplinas de los lenguajes de programación y la administración de bases de datos tienen ciertamente mucho en común, también difieren en determinados aspectos importantes (por supuesto). Para ser específicos:

- Un programa de aplicación está hecho —por definición— para resolver algún problema es pecífico.
- Por el contrario, una base de datos está hecha —de nuevo por definición— para resolver una variedad de problemas diferentes, y algunos de ellos ni siquiera son conocidos al mo mento de establecer la base de datos.

Por lo tanto, en el ambiente de la programación de aplicaciones, incrustar mucha "inteligencia" en objetos complejos es claramente una buena idea: reduce la cantidad de código que es necesario escribir para utilizar esos objetos, mejora la productividad del programador, facilita el mantenimiento de la aplicación, etcétera. Por el contrario, en el ambiente de base de datos, incrustar mucha inteligencia en la base de datos puede o no ser una buena idea: puede simplificar algunos problemas, pero al mismo tiempo puede también hacer que otros sean más difíciles o hasta imposibles de resolver.

(A propósito, éste es exactamente el mismo argumento que se proporcionó en contra de los sistemas de bases de datos prerrelacionales, como el IMS en los años setenta. Un objeto departamento que contiene un conjunto de objetos empleado es, conceptualmente, muy similar a una jerarquía IMS en la cual los "segmentos padre" de departamento tienen "segmentos hijo" de empleados subordinados. Tal jerarquía es adecuada para problemas como "obtener los empleados que trabajan en el departamento de contabilidad". Pero no es adecuada para problemas como "obtener los departamentos que emplean licenciados en Administración de Empresas". Por lo tanto, muchos de los argumentos que se dieron en contra del enfoque jerárquico en los años setenta están volviendo a aparecer, en una forma ligeramente diferente, en el contexto de los objetos.)

A pesar de los comentarios anteriores, mucha gente cree que los sistemas de objetos representan un gran salto hacia adelante en la tecnología de base de datos. En particular, creen que las técnicas de objetos son el enfoque a escoger para las áreas de aplicaciones "complejas", como las siguientes:

- CAD/CAM (Diseño y Manufactura Asistidos por Computadora);
- CIM (Manufactura Integrada por Computadora);
- CASE (Ingeniería de Software Asistida por Computadora);
- GIS (Sistemas de Información Geográfica);
- Ciencia y medicina;
- Almacenamiento y recuperación de documentos;

y así sucesivamente (observe que todas éstas son áreas en las que los productos SQL clásicos tienden a incurrir en problemas). Sin embargo, a lo largo de los años han sido publicados muchos artículos técnicos sobre estos temas y más recientemente, han entrado al mercado varios productos comerciales.

Por lo tanto, en este capítulo daremos un vistazo a todo lo que es la tecnología de bases de datos de objetos. Nuestro propósito es presentar los conceptos más importantes del enfoque

de objetos y, en particular, describir esos conceptos **desde una perspectiva de base de datos** (por el contrario, gran parte de la literatura presenta las ideas desde una perspectiva de la *programación).* La estructura del capítulo es la siguiente. En la siguiente subsección presentamos un ejemplo motivador; un ejemplo que los productos SQL tradicionales no manejan muy bien y por lo tanto, uno en el que la tecnología de objetos tiene una buena oportunidad de hacer un mejor papel. Luego, la sección 24.2 presenta una panorámica de los *objetos, clases, mensajes* y *métodos,* mientras que la sección 24.3 se concentras en determinados aspectos específicos de estos conceptos y los trata a profundidad. La sección 24.4 presenta un ejemplo "de inicio a fin". Después, la sección 24.5 trata unos cuantos temas diversos y la sección 24.6 presenta un resumen.

Un último comentario preliminar: A pesar del hecho de que los sistemas de objetos fueron pensados originalmente para aplicaciones "complejas" como CAD/CAM, por razones obvias basamos nuestros ejemplos en aplicaciones mucho más simples (departamentos y empleados, etcétera). Por supuesto, esta simplificación no invalida de ninguna forma la presentación (y en cualquier caso, si las bases de datos de objetos quieren ser dignas de reconocimiento, también deberán ser capaces de manejar aplicaciones "simples").

Un ejemplo motivador

En esta subsección presentamos un ejemplo simple —creado originalmente por Stonebraker y ampliado por mí en la referencia [24.17]— que ilustra algunos de los problemas que hay con los productos SQL convencionales. La base de datos (que puede ser vista como una aproximación muy simplificada de una base de datos CAD/CAM) hace referencia a *rectángulos,* los cuales por razones de simplicidad, damos por hecho que son "cuadrados" sobre los ejes *X* y *Y;* es decir, todos sus lados son verticales u horizontales. Por lo tanto, cualquier rectángulo individual puede ser unívocamente identificado por las coordenadas *(xl,yl)* y *{x2,y2)* de sus esquinas inferior izquierda y superior derecha, respectivamente (vea la figura 24.1). En SQL:

```
CREATE TABLE RECTÁNGULO
     ( X1 ... , Y1 ... , X2 ... ,  Y2
     UNIQUE ( X1, Y1, X2, Y2 ) )
```

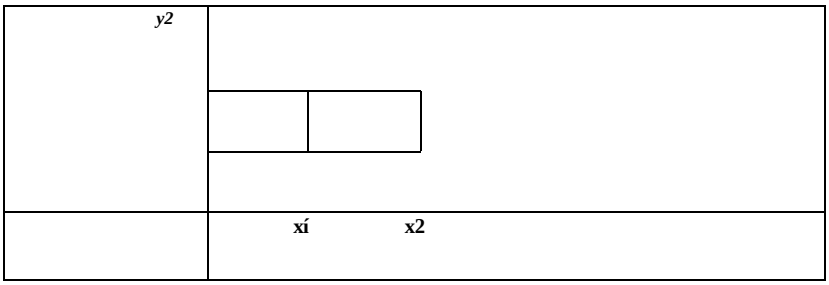


Figura 24.1 El rectángulo *(xl, yl, x2, yl).*

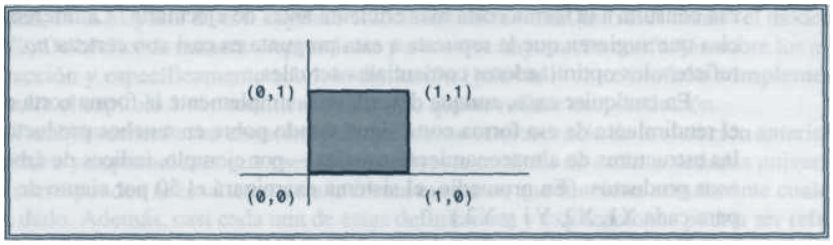


Figura 24.2 El cuadrado unitario (0, 0,1,1).

Ahora considere la consulta "obtener todos los rectángulos que traslapen al cuadrado unitario (0, 0, 1, 1)" (vea la figura 24.2). La formulación "obvia" de esta consulta es:

```
SELEC
FROM   RECTÁNGULO
WHERE  ( X1 >- 0 AN X <  1 AND Y1 >- 0 AND Y1 <= 1 )
                     — l  esquina inferior izquierda dentro del cuadrado unitario
OR     ( X2 >= 0 AN X o  1 AND Y2 >» 0 AND Y2 <= 1 )
                     l  esquina superior derecha dentro del cuadrado unitario
OR     ( X1 >= 0 AN X <  1 AND Y2 >= 0 AND Y2 <= 1 )
                     — l  esquina superior izquierda dentro del cuadrado unitario
OR     ( X2 >= 0 AN X <  1 AND Y1 >= 0 AND Y1 <= 1 )
                     — l  esquina inferior derecha dentro del cuadrado unitario
OR     ( X1 <- 0 AN X >  1 AND Y1 <= 0 AND Y2 >= 1 )
                       e  rectángulo incluye por completo al cuadrado unitario
OR     ( X1 <= 0 AN X >  1 AND Y1 >= 0 AND Y1 <= 1 )
                     — e  borde inferior cruza el cuadrado unitario
OR     ( X1 >= 0 AN X <  1 AND Y1 <= 0 AND Y2 >= 1 )
                     — e  borde izquierdo cruza el cuadrado unitario
OR     ( X2 >= 0 AN X <  1 AND Y1 <• 0 AND Y2 >= 1 )
                     — e  borde derecho cruza el cuadrado unitario
OR     ( X1 <= 0 AN X >  1 AND Y2 >= 0 AND Y2 <= 1 ) ;
                       e  borde superior cruza el cuadrado unitario
```

(Ejercicio: convénzase usted mismo de que esta formulación es correcta.)

Sin embargo, pensándolo un poco más, podemos ver que la consulta puede ser expresada de una forma más simple como:

```
SELECT ...
FROM  RECTÁNGULO
WHERE  ( XI <« 1 AND Y1 <■ 1
                —  la esquina inferior izquierda está "hacia abajo" de (1,1)
AND     X2 >= 0 AND Y2 >= 0 ) ;
                —  la esquina superior derecha está "hacia arriba" de (0,0)
```

(El ejercicio 24.3 al final del capítulo le pide que se convenza usted mismo de que también esta formulación es correcta.)

Ahora la pregunta es: ¿podría el optimizador del sistema transformar la forma larga original de la consulta en la forma corta correspondiente? En otras palabras, suponga que el usuario expresa la consulta en la forma larga "obvia" y también ineficiente; ¿podría el optimizador reducir

J

esa consulta a la forma corta más eficiente antes de ejecutarla? La referencia [24.17] da evidencias que sugieren que la repuesta a esta pregunta es casi con certeza *no,* al menos en lo que se refiere a los optimizadores comerciales actuales.

En cualquier caso, aunque describimos simplemente la forma corta como "más eficiente", el rendimiento de esa forma corta sigue siendo pobre en muchos productos, tomando en cuenta las estructuras de almacenamiento usuales —por ejemplo, índices de árbol B— soportadas por esos productos. (En promedio, el sistema examinará el 50 por ciento de las entradas de índice paracadaXl,X2,YlyY2.)

Por lo tanto, vemos que los productos SQL convencionales también son inadecuados en ciertos aspectos. Para ser más específicos, problemas como el del rectángulo muestran claramente que determinadas solicitudes "simples" de usuario (a) son irrazonablemente difíciles de expresar y (b) se ejecutan con un rendimiento inaceptablemente bajo para esos productos. Dichas consideraciones proporcionan gran parte de la motivación que hay tras el interés actual en los sistemas de objetos.

Nota: Daremos una "buena" solución al problema de los rectángulos en el capítulo 25 (sección 25.1).

24.2 OBJETOS, CLASES, MÉTODOS Y MENSAJES

En esta sección presentamos algunos de los conceptos principales del enfoque de objetos; es decir, los propios *objetos* (por supuesto), las *clases de objetos,* los *métodos* y los *mensajes.* También, cada vez que sea posible o adecuado, relacionamos estos conceptos con otros más familiares. De hecho, probablemente sea útil mostrar en este momento una correspondencia burda entre los términos de objetos y la terminología tradicional (vea la figura 24.3).

Advertencia: Antes de entrar en detalles, debemos prevenirle que en el mundo de los objetos usted no encontrará el tipo de precisión al que está (o debería estar) acostumbrado en el mundo relacional. Además, muchos conceptos de objetos —o las definiciones publicadas de esos objetos— son bastante imprecisos y hay muy poco consenso verdadero y mucho desacuerdo, incluso en el nivel más básico (como lo mostrará una lectura cuidadosa de, por ejemplo, las referencias [24.11], [24.48] y [24.52]). En particular, no hay un "modelo de datos de objetos" abstracto ni formalmente definido, y tampoco hay consenso sobre un modelo informal. (Por tales

Término de objetos	*Término tradicional*
Objeto inmutable	Valor Variable Tipo
Objeto mutable	Operador Invocación de operador
Clase de objetos	
Método Mensaje	

Figura 24.3 Terminología de objetos (resumen).

razones, en la mayoría de este capítulo colocamos entre comillas frases como "el modelo de objetos".) De hecho, de manera sorprendente parece que hay mucha confusión sobre los niveles de abstracción y específicamente sobre la distinción (¡crucial!) entre **modelo** e **implementation.** Consulte el capítulo 1 si necesita recordar a lo que se refiere esta distinción.

También deberá estar consciente de que, como consecuencia de la situación anterior, las definiciones y explicaciones que presentamos en este capítulo *no* están acordadas universalmente y *no* corresponden necesariamente a la forma exacta en que funciona actualmente cualquier sistema dado. Además, casi cada una de estas definiciones y explicaciones podría ser refutada por algún otro escritor en este campo (y probablemente lo será).

Una panorámica de la tecnología de objetos

Pregunta: ¿Qué es un objeto? *Respuesta:* ¡Todo!

Es un principio básico del enfoque de objetos que **"todo es un objeto"** (a veces "todo es un objeto de **primera clase").** Algunos objetos son **inmutables;** ejemplos de esto pueden ser los enteros (por ejemplo, 3 y 42) y las cadenas de caracteres (por ejemplo, "Mozart", "¡Hayduke vive!"). Otros objetos son **mutables;** algunos ejemplos podrían ser los objetos de departamento y empleado que mencionamos al inicio de la sección 24.1. Por lo tanto, en la terminología tradicional, los objetos inmutables corresponden a los *valores* y los objetos mutables corresponden a las *variables;** donde los valores y variables en cuestión pueden ser de una complejidad cualquiera (es decir, pueden utilizar cualquiera o todos los tipos y generadores de tipos usuales de los lenguajes de programación: números, cadenas, listas, arreglos, pilas, etcétera). *Nota:* En algunos sistemas el término *objeto* está reservado únicamente para el caso mutable (y entonces, se usa el término *valor*—o a veces *literal*— para el caso inmutable). Incluso en aquellos sistemas, donde el término "objeto" se refiere estrictamente a ambos casos, habrá que estar consciente de que es común que en contextos informales se tome a este término para referirse específicamente a un objeto mutable, mientras no se diga explícitamente lo contrario.

Todo objeto tiene un *tipo* (el término en objetos es **clase).** A los objetos individuales con frecuencia se les llama específicamente **ejemplares (instancias)** de objeto, para distinguirlos con claridad del tipo o clase del objeto correspondiente. También observe que aquí estamos usando el término *tipo* en su sentido usual de lenguaje de programación (como en el capítulo 5); por lo tanto, tomamos en particular a este término para que incluya el conjunto de *operadores* (el término en el entorno de objetos es **métodos)** que pueden ser aplicados a los objetos de ese tipo. *Nota:* De hecho, algunos sistemas de objetos distinguen entre tipos y clases; trataremos tales sistemas brevemente en la sección 24.3, pero mientras no lleguemos a ese punto, usaremos los términos en forma intercambiable.

Los objetos están **encapsulados.** Esto significa que la representación física —es decir, la estructura interna— de un objeto de estos, digamos un objeto DEPTO ("departamento"), no es visible para los usuarios de ese objeto; en vez de ello, los usuarios sólo saben que el objeto es capaz

---

*Sin embargo, observe que el término *variable* sin calificativos se usa frecuentemente en los contextos de objetos para referirse muy específicamente a una variable que guarda un *ID de objeto* (vea más adelante en esta sección).

de ejecutar determinadas operaciones (métodos).* Por ejemplo, los métodos que se aplican a los objetos DEPTO pueden ser CONTRATAR_EMP, DESPEDIR_EMP, RECORTAR_PRESUPUESTO, etcétera. *Observe cuidadosamente que tales métodos constituyen las ÚNICAS operaciones que pueden ser aplicadas a los objetos en cuestión.* Está permitido que el código que implementa esos métodos vea la representación interna de los objetos —en este vocabulario, a ese código (pero sólo a ése) se le permite "romper la encapsulación"[†]— pero por supuesto, es probable que ese código tampoco sea visible ante los usuarios.

La ventaja de la encapsulación es que permite que la representación interna de los objetos sea cambiada sin necesidad de que las aplicaciones que los usan tengan que ser reescritas (siempre y cuando, por supuesto, cualquiera de esos cambios en la representación interna esté acompañado por un cambio correspondiente en el código que implementa los métodos aplicables). En otras palabras, la encapsulación implica la **independencia** tísica **de los datos.**

Ahora bien, la caracterización anterior de la encapsulación —en términos de la independencia de datos— tiene sentido desde una perspectiva de una base de datos, pero por lo general ésta no es la forma en que el concepto se describe en la literatura de objetos. En su lugar, los objetos encapsulados son descritos como si tuvieran una *memoria privada* y una *interfaz pública:*

- La **memoria privada** consiste en **variables de ejemplar** (también conocidas como *miembros o atributos),* cuyos valores representan el estado interno del objeto. Ahora bien, en un sistema "puro" las variables de ejemplar son completamente privadas y ocultas ante los usuarios, aunque (como dijimos anteriormente) son visibles ante el código que implementa los método dos. Sin embargo, muchos sistemas *no* son puros en este sentido, sino que exponen las varia bles de ejemplar ante los usuarios, un punto que retomaremos en la siguiente subsección.
- La **interfaz pública** consiste en definiciones de interfaz para los métodos aplicados a este objeto. Esas definiciones de interfaz corresponden a lo que en el capítulo 19 llamamos *sig naturas de especificación;* salvo que (como explicamos en el siguiente párrafo) los sistemas de objetos insisten, por lo general, en que tales signaturas están atadas a un solo tipo o clase "de destino" específico y, en cambio, no teníamos tal noción en el capítulo 19 (y tampoco la encontramos necesaria, ni incluso deseable [3.3]). Como ya mencionamos, el código que implementa esos métodos, al igual que las variables de ejemplar, está oculto ante el usuario.

---

*En la literatura hay mucha confusión alrededor de la noción de encapsulación. La posición que parece tener más sentido, y la que adoptamos en este libro, es que un objeto está encapsulado si y sólo si es **escalar** en el sentido del capítulo 5 (es decir, si y sólo si no tiene componentes visibles para el usuario); por lo tanto, *encapsulado* y *escalar* significan exactamente lo mismo. Observe que determinados objetos de "colección" —vea la sección 24.3—no son en definitiva escalares y por lo tanto, de acuerdo con esta definición, tampoco encapsulados. Por el contrario, algunos escritores establecen categóricamente que *todos* los objetos están encapsulados, una posición que inevitablemente conduce a determinadas contradicciones. Mientras que otros toman el concepto para dar a entender que, además de que la estructura interna está oculta, *los métodos correspondientes están unidos físicamente con* (es decir, son físicamente parte de) el objeto o clase de objeto en cuestión. Sentimos que esta última interpretación combina las consideraciones de modelo e implementación; de hecho, esa confusión es otra de las razones por las cuales, como dijimos en el capítulo 5, preferimos no usar en absoluto el término "encapsulación". Sin embargo, en el presente capítulo tenemos que usarlo de vez en cuando.

[†]Nosotros mismos recomendamos una disciplina más fuerte [3.3]. Sólo debería permitirse que los selectores y los operadores THE_ —vea el capítulo 5— rompieran la encapsulación en este sentido; todos lo demás operadores deberían ser implementados en términos de esos selectores y de los operadores THE_. En otras palabras, ¡codifique a la defensiva! Sin embargo, por lo general los sistemas de objetos no proporcionan a los operadores THE_ como tales, sino que en su lugar proporcionan operadores de "obtener y colocar" (vea la sección 24.4) que *no* son una contraparte exacta [24.22].

Nota: Sería más preciso decir que la interfaz pública es parte del **objeto de** definición **de clase** para los objetos en cuestión, en lugar de decir que es parte del objeto individual mismo. (Después de todo, la interfaz pública es común para todos los objetos de la clase en cuestión, en lugar de ser específica de algún objeto individual.) El objeto de definición de clase, u ODC, es el objeto que define la clase de la cual el objeto en cuestión es un ejemplar; es similar a una entrada de catálogo en un sistema relacional convencional.

Los métodos son llamados por medio de **mensajes.** Un mensaje es esencialmente sólo una *invocación de operador* en la cual se distingue un argumento, el **destino,** y se le da un tratamiento sintáctico especial. Por ejemplo, el siguiente podría ser un mensaje para el departamento D solicitando la contratación del empleado E:

```
D CONTRATA_EMP ( E )
```

(sintaxis hipotética; vea la subsección "Clase contra ejemplar contra colección" en la sección 24.3, para una explicación de los argumentos D y E). Aquí, el destino es el objeto departamento indicado por D. La analogía de este mensaje en un lenguaje de programación más convencional (es decir, uno que trate en forma igual a todos sus argumentos) podría verse como sigue:*

```
CONTRATA_EMP ( D, E )
```

En la práctica, un sistema de objetos vendrá equipado con varias clases y métodos *integrados.* En particular, es casi seguro que el sistema proporcionará clases como NUMERIC (con métodos "=", "<", "+", "-", etcétera), CHAR (con métodos "=", "<", "II", SUBSTR, etcétera), entre otras. Por supuesto, el sistema también brindará facilidades para que los usuarios debidamente capacitados definan e implementen clases y métodos propios.

Variables de ejemplar

Ahora daremos una vista un poco más cercana al concepto de variables de ejemplar (el hecho es que este tema está rodeado de cierta confusión). Como dijimos anteriormente, en un sistema puro las variables de ejemplar están ocultas ante el usuario. Sin embargo, por desgracia la mayoría de los sistemas no son puros en este sentido. Por consecuencia, en la práctica es necesario distinguir entre variables de ejemplar **públicas y privadas,** donde las privadas están verdaderamente ocultas y las públicas no.

A manera de ejemplo, suponga que tenemos una clase de objetos de *segmentos de línea* y suponga que los segmentos de línea están representados físicamente por sus puntos INICIO Y FIN. Entonces, el sistema permitirá generalmente que los usuarios escriban expresiones de la forma í/.INICIO y J/.FIN para "obtener" los puntos de INICIO y FIN de un segmento de línea *si* dado. Por lo tanto, INICIO y FIN son variables de ejemplar *públicas* (observe que, por definición, el acceso a las variables de ejemplar públicas debe ser por medio de alguna sintaxis especial; por lo general la calificación por punto, como lo sugiere nuestro ejemplo). Y si es cambiada la representación física del segmento de línea, digamos a la combinación de PUNTO_MEDIO, LONGITUD

Tratar un argumento como especial puede facilitar al sistema la realización del proceso de *enlace en tiempo de ejecución* que describimos en el capítulo 19. Sin embargo, tiene muchas desventajas (vea la referencia [3.3]), y la no menos importante es que puede dificultar la escritura del código de implementación a quien ¡mplementa el método. Otra es que el argumento elegido como destino (en los casos donde hay alternativa, es decir, cada vez que hay dos o más argumentos) es, por supuesto, arbitrario.

y PENDIENTE, entonces ningún programa que incluya expresiones como s/.INICIO y s/.FIN, funcionará. En otras palabras, hemos perdido la independencia de datos.

Observe ahora que las variables de ejemplar públicas son lógicamente innecesarias. Suponga que definimos los métodos OBTENER JNICIO, OBTENER_FIN, OBTENER_PUNTO_MEDI0, OBTENER_LONGITUD y OBTENER_PENDIENTE para los segmentos de línea. Entonces el usuario podrá "obtener" el punto inicial, el punto final, el punto medio y así sucesivamente, para el segmento de línea *si* por medio de *invocaciones a métodos* adecuadas, como OBTENERJNICIO(s/), OBTENER_FIN(s/), OBTENER_PUNTO_MEDIO(s/) y así sucesivamente. ¡Y ahora no importa cuál es la representación física del segmento de línea!, siempre y cuando los diversos métodos OBTENER_ sean implementados adecuadamente (o reimplementados si cambia la representación física). Además, no habría nada de malo en permitir que el usuario reduzca, por ejemplo, OBTENER_INICIO(i/) a simplemente s/.INICIO **como abreviatura;** observe que la disponibilidad de esta abreviatura *no* haría que INICIO fuera una variable de ejemplar pública. Sin embargo, desgraciadamente los sistemas reales no operan de esta forma; en general, las variables de ejemplar públicas en realidad exponen la representación física (o parte de ella, aunque puede haber algunas variables de ejemplar adicionales que ciertamente son privadas y ocultas). Por lo tanto, de acuerdo con la práctica común, daremos por hecho a partir de este punto (mientras no digamos otra cosa) que los objetos exponen generalmente determinadas variables de ejemplar públicas, aunque el concepto sea lógicamente innecesario.

Hay un punto relacionado que también debemos tratar aquí. Suponga que determinados argumentos —que el usuario puede ver, en términos generales, como argumentos de "variable de ejemplar"— son necesarios para crear objetos de una clase en particular.* Entonces *no* podemos deducir que esas mismas "variables de ejemplar" pueden ser usadas para cualquier propósito. Por ejemplo, suponga que la creación de un segmento de línea requiere que especifiquemos los puntos INICIO y FIN aplicables. Entonces no es lógico que podamos obtener, por ejemplo, todos los segmentos de línea con un punto de INICIO dado; más bien, dicha solicitud será válida sólo si se ha definido un método adecuado.

Por último, observe que algunos sistemas soportan una variación sobre las variables de ejemplar privadas, llamada variables de ejemplar **protegidas.** Si los objetos de la clase C tienen una variable de ejemplar protegida *P,* entonces *P* es visible para el código que implementa los métodos definidos para la clase *C* (por supuesto) *y* para el código que implementa los métodos definidos para *cualquier subclase* (a cualquier nivel) de la clase C. Vea el final de la sección 24.3 para una breve explicación de las subclases.

Identidad de objetos

Todo objeto tiene un *identificador* único llamado su **ID de objeto,** u OÍD. Los objetos inmutables, como el entero 42, son *autoidentificables;* es decir, sirven como su propio OÍD. Por el contrario, los objetos mutables tienen *direcciones* (conceptuales) como OIDs, y esas direcciones pueden ser usadas en cualquier lugar de la base de datos como *apuntadores* (conceptuales) para referirse a los objetos en cuestión. Tales direcciones no son expuestas directamente ante los usuarios, pero (por ejemplo) pueden ser asignadas a variables de programa y a variables de ejemplar dentro de otros objetos. Vea las secciones 24.3 y 24.4 para comentarios adicionales.

---

*A propósito, los objetos en cuestión deben necesariamente ser mutables (¿por qué?).

De paso, hacemos notar que en ocasiones se ve como una ventaja de los sistemas de objetos el hecho que dos objetos distintos puedan ser idénticos en todos los aspectos visibles por el usuario —por ejemplo, que sean duplicados uno del otro— y sigan siendo distinguibles por sus OIDs. Sin embargo, en mi opinión esta afirmación parece engañosa. ¿Cómo podría un *usuario* distinguir externamente entre dos de estos objetos? Vea las referencias [5.3], [5.6] y (en particular) [24.19] para mayores comentarios sobre este tema.

24.3 UNA MIRADA MÁS CERCANA

Daremos ahora una mirada más cercana a algunas de las ideas que presentamos en la sección anterior. Suponga que queremos definir dos clases de objetos, DEPTO (departamentos) y EMP (empleados). Suponga también que las clases DINERO y TRABAJO ya han sido definidas por el usuario y que la clase CHAR está integrada. Entonces las definiciones de clase necesarias para DEPTO y EMP pueden verse de la siguiente forma (sintaxis hipotética):

```
CLASS DEPTO
  PUBLIC      ( DEPTO#          CHAR,
                NOMDEPTO        CHAR,
                PRESUPUESTO     DINERO
                GER             REF (   EMP ),
                EMPS            REF (   SET ( REF ( EMP ) )
  METHODS       ( CONTRATAR     ( REF (  EMP ) )    . código
                DESPEDIREMP     ( REF (  EMP ) ) .  .código
CLASS EMP
  PUBLIC      ( EMP#            CHAR,
                NOMEMP          CHAR,
                SALARIO         DINERO
                POSICIÓN        REF (   TRABAJO ) ) . ■ .
                      METHODS   ( ... )  ... ;
```

Surgen estos puntos:

Hemos decidido representar a los departamentos y a los empleados por medio de una **jerarquía de contención,** en la cual los objetos EMP están contenidos conceptualmente dentro de objetos DEPTO. Por lo tanto, los objetos de la clase DEPTO incluyen una variable de ejemplar pública —llamada GER— que representa al gerente del departamento dado, así como otra variable llamada EMPS que representa a los empleados del departamento dado. Para ser más precisos, los objetos de la clase DEPTO incluyen una variable de ejemplar pública llamada GER cuyo valor es una *referencia* ("REF") hacia un empleado, y otra llamada EMPS cuyo valor es una referencia hacia un conjunto de referencias a empleados. *(Nota:* Aquí "referencia a" en realidad significa *OÍD de;* vea a continuación un comentario adicional.) Dentro de unos momentos ampliaremos la noción de jerarquía de contención.

Para efectos del ejemplo, hemos decidido no incluir una variable de ejemplar dentro de objetos de la clase EMP cuyo valor sea un OÍD de departamento o un valor DEPTO# (una variable de ejemplar de "clave externa"). Esta decisión es consistente con nuestra decisión de representar a los departamentos y empleados por medio de una jerarquía de contención. Sin embargo, esto significa que no hay una forma directa para obtener, a partir de un objeto EMP dado, el objeto DEPTO correspondiente. Vea la sección 24.5, subsección "Vínculos", para una explicación más amplia de este punto.

3. Observe que cada definición de clase incluye definiciones (omitimos los detalles de codificación) de los métodos aplicados a los objetos de la clase en cuestión. La clase de destino para tales métodos es, por supuesto, la clase cuya definición incluye la definición de los métodos en cuestión.*

La figura 24.4 muestra algunos ejemplares de objeto correspondientes a las clases DEPTO y EMP que acabamos de definir. Primero considere el objeto EMP que está en la parte superior de esa figura (con un OÍD *eee),* el cual contiene:

- Un objeto inmutable "E001" de la clase integrada CHAR en su variable de ejemplar pública EMP#.
- Un objeto inmutable "Smith" de la clase integrada CHAR en su variable de ejemplar pública NOMEMP.
- Un objeto inmutable "$50,000" de la clase definida por el usuario DINERO en su variable de ejemplar pública SALARIO.
- El **OÍD** de un objeto mutable de la clase definida por el usuario TRABAJO en su variable de ejemplar pública POSICIÓN.

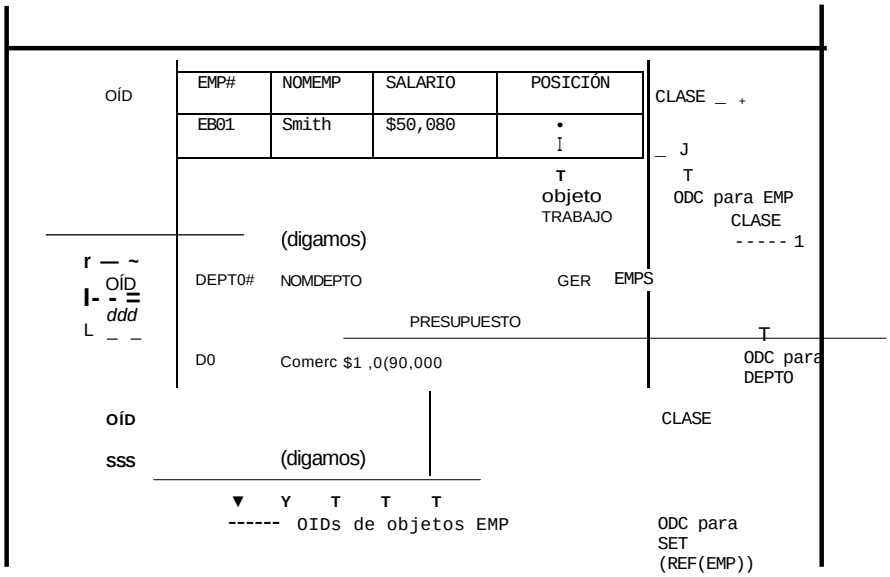


Figura 24.4 Ejemplares de ejemplo de DEPTO y EMP.

*Observe que nuestra sintaxis hipotética (innecesaria, pero muy común) combina consideraciones de modelo y de implementación. Observe también que hemos argumentado en muchos lugares [13.11] que ¡los departamentos y empleados son, de cualquier forma, malos ejemplos de clases de objetos! Sin embargo, una mayor explicación de este punto particular nos apartaría demasiado del tema.

También contiene al menos dos variables de ejemplar privadas adicionales, una que contiene el OÍD *eee* del propio objeto EMP y una que contiene el OÍD del objeto que define a la clase —es decir, el ODC— para EMPs (con el cual la implementación podrá encontrar la información del descriptor para este objeto). *Nota:* Estos dos OIDs pueden estar, o no, almacenados físicamente dentro del objeto. Por ejemplo, el valor *eee* no debe necesariamente ser guardado como parte del objeto EMP relevante; sólo es necesario que la implementación tenga alguna forma de localizar ese objeto EMP a partir del valor *eee* (es decir, alguna forma de correspondencia entre ese valor *eee* con la dirección física del objeto EMP). Pero de manera conceptual, el usuario siempre podrá ver al OÍD como si fuera parte del objeto, como se muestra.

Ahora pasamos al objeto DEPTO, que está en el centro de la figura, con el OÍD *ddd.* Ese objeto contiene:

- Un objeto inmutable "D01" de la clase integrada CHAR en su variable de ejemplar pública DEPTO*.
- Un objeto inmutable "Comerc" de la clase integrada CHAR en su variable de ejemplar pública NOMDEPTO.
- Un objeto inmutable "$ 1,000,000" de la clase definida por el usuario DINERO en su varia ble de ejemplar pública PRESUPUESTO.
- El OÍD *eee* de un objeto mutable de la clase definida por el usuario EMP en su variable de ejemplar pública GER (éste es el OÍD del objeto que representa al gerente del departamento).
- El OÍD *sss* de un objeto mutable de la clase definida por el usuario SET(REF(EMP)) en su variable de ejemplar pública EMPS (vea más adelante).
- Dos variables de ejemplar privadas que contienen, respectivamente, el OÍD *ddd* del propio objeto DEPTO y el OÍD del objeto de definición de clase correspondiente.

Y el objeto con el OÍD m contiene un conjunto de OIDs de objetos EMP individuales (mutables), además de las variables de ejemplar privadas usuales.

Ahora bien, la figura 24.4 muestra los objetos "como son realmente"; es decir, la figura ilustra el componente de la *estructura de datos* del "modelo de objetos", y tales figuras deben ser comprendidas claramente por los usuarios de ese modelo. Sin embargo, los textos y presentaciones de objetos no muestran, por lo general, diagramas como el de la figura 24.4; en vez de ello, representan generalmente la situación como muestra la figura 24.5 que aparece más adelante (la cual puede ser considerada como que está en un nivel de abstracción más alto y por lo tanto, es más fácil de entender).

La representación que muestra la figura 24.5 es ciertamente más consistente con la interpretación de jerarquía de contención. Sin embargo, opaca el hecho más importante (que ya enfatizamos anteriormente) de que los objetos a menudo no contienen otros objetos como tales, sino que en vez de ello, contienen a los **OIDs** de —es decir, apuntadores hacia— otros objetos. Por ejemplo, la figura 24.5 sugiere claramente que el objeto DEPTO para el departamento D01, incluye *dos veces* al objeto EMP para el empleado E001 (lo que implica, entre otras cosas, que el empleado E001 puede ser representado en forma inconsistente como si tuviera dos salarios diferentes en sus dos apariciones diferentes). Esta prestidigitación es el origen de mucha confusión y es la razón por la cual preferimos ejemplos como el de la figura 24.4.

Además, observamos que las definiciones de clases de objetos auténticas incrementan frecuentemente la confusión, debido a que no definen variables de ejemplar como "REFs" (como lo hace nuestra sintaxis hipotética) sino que, en vez de ello, reflejan directamente la interpretación de

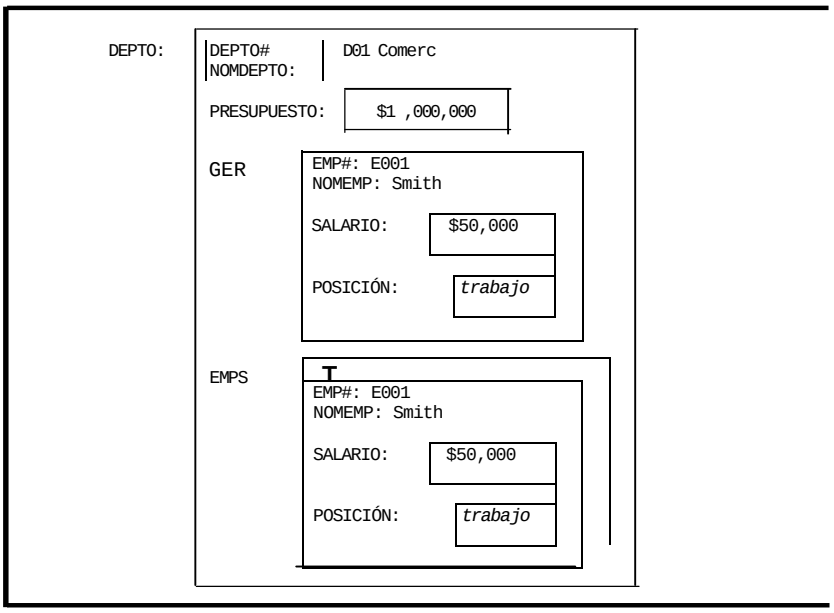


Figura 24.5 Ejemplares de DEPTO y EMP como una jerarquía de contención.

jerarquía de contención. Por lo tanto, por ejemplo, la variable de ejemplar EMPS en la clase DEPTO podría ser definida no como REF(SET(REF(EMP))), sino simplemente como SET(EMP). Aunque es algo latoso, preferimos nuestro estilo de definición por su claridad y precisión.

También, vale la pena señalar que todas las viejas críticas sobre las jerarquías en general, como se encuentran por ejemplo en el IMS, se aplican en particular a las jerarquías de contención. Aquí el espacio no nos permite consideraciones detalladas sobre esas críticas; pero basta decir que el aspecto primordial es la **falta de simetría.** En particular, las jerarquías por sí mismas, no conducen bien a la representación de los vínculos muchos a muchos. Considere por ejemplo a los proveedores y partes: ¿los proveedores contienen a las partes o *viceversa*[1]*?* ¿o ambos? ¿qué hay acerca de los proveedores, las partes y los proyectos?

De hecho, el asunto se pone más confuso que como habíamos sugerido anteriormente. Por un lado, decimos (tal como explicamos anteriormente) que los objetos son *jerarquías,* lo que significa que están sujetos a las críticas usuales sobre las jerarquías, como ya dijimos. Sin embargo, por otro lado, queda claro —a partir de figuras como la 24.4— que los objetos en realidad no son jerarquías sino **tupias,** donde los componentes de la tupia pueden ser cualesquiera de los siguientes:

1. "Subobjetos" inmutables (es decir, valores autoidentificados, tales como enteros o cantida des de dinero).
2. OIDs de "subobjetos" mutables (es decir, referencias o apuntadores hacia otros objetos mu tables, posiblemente compartidos).
3. Conjuntos, listas, arreglos,... de 1., 2. o 3.

(Más determinados componentes ocultos OÍD, **OÍD** de ODC, etcétera.) Observe en particular el punto 3: los sistemas de objetos soportan generalmente varios **generadores de tipo** de "colección" —por ejemplo, SET, LIST, ARRAY, BAG, etcétera (aunque, por lo general, ¡no existe RELATION!)— y esos generadores pueden ser combinados en formas arbitrarias. Por ejemplo, un arreglo de listas de bolsas que contienen arreglos de apuntadores hacia variables enteras, puede constituir un solo objeto mutable en circunstancias adecuadas. Vea la subsección Clase contra Ejemplar contra Colección, para una mejor explicación.

Revisión de los IDs de objetos

Por lo general, los DBMSs relaciónales se apoyan en claves definidas y controladas por el usuario —"claves de usuario", para abreviar— para efectos de identificación y referencia de entidades (en realidad, los apuntadores estilo OÍD están expresamente prohibidos en las bases de datos relaciónales, como sabemos por el capítulo 3). Sin embargo, es bien sabido que las claves de usuario padecen varios problemas; las referencias [13.10] y [13.16] tratan tales problemas en detalle y argumentan que los DBMSs relaciónales deben soportar, en su lugar, claves definidas por el *sistema* ("sustitutos"), al menos como una opción. Y los argumentos en favor de los OIDs en los sistemas de objetos son similares (de alguna forma) a los argumentos a favor de los sustitutos en los sistemas relaciónales. (Sin embargo, no caiga en el error de igualar a los dos: los sustitutos son *valores* y son visibles ante el usuario; en cambio los OIDs son *direcciones* —al menos conceptualmente— y están ocultos ante el usuario. Vea la referencia [24.19] para una amplia exposición de estas distinciones y asuntos relacionados.) Surgen los siguientes puntos y preguntas:

1. Primero, observe que los OIDs no evitan la necesidad de claves de usuario, como veremos en la sección 24.4. Para ser más precisos, las claves de usuario todavía son necesarias para interactuar con el mundo externo, aunque todas las referencias a objetos usadas dentro de la base de datos se realicen por medio de OIDs.
2. ¿Cuál es el OÍD de un objeto *derivado!* —por ejemplo, ¿la "junta" de un EMP dado y su DEPTO correspondiente, o la "proyección" de un DEPTO dado sobre PRESUPUESTO y GER? (A propósito, este asunto de los *objetos derivados* es importante, pero tenemos que dejarlo por ahora. Vea la sección 24.5.)
3. Los OIDs son la fuente de las críticas que se oyen frecuentemente con respecto a que los sistemas de objetos son vistos como un "CODASYL calentado", donde (como indicamos en el capítulo 1) el término CODASYL se refiere genéricamente a determinados sistemas de bases de datos de red (es decir, prerrelacionales) tales como IDMS. Ciertamente, los OIDs tienden a conducir a un estilo de programación de bajo nivel y de persecución de apunta dores (vea la sección 24.4) que recuerda mucho el viejo estilo CODASYL. También, el hecho de que los OIDs sean apuntadores, cuenta para las reclamaciones que también se oyen ocasionalmente en relación de que:
   - Los sistemas CODASYL están más cercanos a los sistemas de objetos de lo que lo que están los sistemas relaciónales.
   - Los sistemas relaciónales están *basados en el valor,* mientras que los sistemas de obje tos están *basados en la identidad.*

Clase contra ejemplar contra colección

Los sistemas de objetos distinguen muy claramente los conceptos de *clase, ejemplar y colección.* Una **clase** es (como ya explicamos) básicamente un tipo de dato, posiblemente integrado, posiblemente definido por el usuario y de una complejidad cualquiera.* Toda clase entiende un mensaje **NEW,** el cual ocasiona que se cree un nuevo **ejemplar** (mutable) de la clase. *(Nota:* Al método llamado por el mensaje NEW se le conoce a veces como *función constructora.)* Por ejemplo (sintaxis hipotética):

```
E := EMP NEW ( 'E001', 'Smith', DINERO ( 50000 ), POS ) ;
```

Aquí, POS es una variable del programa que contiene el OÍD de algún objeto TRABAJO. El método NEW es invocado sobre la clase EMP; crea un nuevo ejemplar de esa clase, lo inicializa a los valores especificados y regresa el OÍD del nuevo ejemplar. Ese OÍD es entonces asignado a la variable de programa E.

Después, debido a que los objetos pueden ser referidos (por medio de su OÍD) por cualquier cantidad de otros objetos, éstos pueden en efecto ser *compartidos* por esos otros objetos. En particular, pueden pertenecer simultáneamente a cualquier número de objetos de **colección.** Para continuar con el ejemplo:

```
CLASS COLEMP

   PUBLIC ( EMPS REF ( SET ( REF ( EMP ) ) ) ) ... ;

TODOSEMPS := COLEMP NEW ( ) ;

TODOS_EMPS AGREGAR ( E ) ;
```

Explicación:

- Un objeto de la clase COL_EMP contiene una sola variable de ejemplar pública, llamada EMPS, cuyo valor es un apuntador (OÍD) hacia un objeto mutable cuyo valor es un con junto de apuntadores (OIDs) hacia objetos EMP individuales.
- TODOS_EMPS es una variable de programa cuyo valor es el OÍD de un objeto de la clase COL_EMP. Después de la operación de asignación, ésta contiene el OÍD de un objeto de éstos cuyo valor es a su vez el OÍD de un conjunto *vacío* de OIDs de objetos EMP.
- AGREGAR es un método que es entendido por objetos de la clase COL_EMP. En el ejem plo, ese método se aplica al objeto de esa clase cuyo OÍD está dado en la variable de progra ma TODOS_EMPS; su efecto es agregar el OÍD del objeto EMP, dado por la variable de programa E, al conjunto (previamente vacío) de OIDs cuyo OÍD (del conjunto) está dado por el objeto COL_EMP que está en la variable de programa TODOS_EMPS.

Después de la secuencia de operaciones anterior, podemos decir (en términos generales) que la variable TODOS_EMPS denota una colección de EMPs que contiene actualmente sólo un EMP, precisamente el empleado E001. (A propósito, ¡observe la necesidad de mencionar un valor de clave de usuario en esta última frase!).

---

*Como mencionamos en la sección 24.2, algunos sistemas utilizan tanto "tipo" como "clase"; en cuyo caso, "tipo" significa *tipo* o *intensión y* "clase" significa *extensión* (es decir, una determinada *colección),* o a veces *implementación* (del tipo en cuestión). Luego, para repetir, otros sistemas usan los términos de forma inversa... Continuaremos tomando a "clase" para que signifique un tipo en el sentido del capítulo 5.

Por supuesto, podemos tener cualquier cantidad de "conjuntos de empleados" distintos, y posiblemente traslapados, en cualquier momento dado:

```
PROGRAMADORES := COLEMP NEW ( ) ;

PROGRAMADORES AGREGAR ( E ) ;


ALTAMENTE_PAGADOS : = COLEMP NEW ( ) ;
ALTAMENTE_PAGADOS AGREGAR ( E ) ;
```

y así sucesivamente. Contraste la situación en los sistemas SQL. Por ejemplo, la declaración SQL

```
CREATE TABLE EMP ( EMP# ... ,
     NOMEMP ... , SALARIO ...
     , POSICIÓN ... ) ... ;
```

crea simultáneamente un tipo *y* una colección; el tipo está definido por el encabezado de la tabla y la colección (vacía en un principio) es el cuerpo de la tabla. De manera similar, la instrucción SQL

```
INSERT INTO EMP ( ... ) VALUES (...);
```

crea una fila EMP individual *y* la agrega simultáneamente a la colección EMP. Por lo tanto, en SQL:

1. No hay forma de que exista un "objeto" EMP individual sin ser parte de alguna "colección"; en realidad, exactamente de una sola "colección" (pero vea más adelante).
2. No hay forma directa para crear dos "colecciones" distintas de la misma "clase" de "obje tos" EMP (pero vea más adelante).
3. No hay forma directa para compartir el mismo "objeto" a través de distintas "colecciones" de "objetos" EMP (pero vea más adelante).

Por lo menos, en ocasiones se oyen los reclamos anteriores. Sin embargo, no soportan de hecho un fuerte escrutinio. Primero, el mecanismo de clave externa puede ser usado para lograr un efecto equivalente en cada caso; por ejemplo, podríamos definir dos tablas base adicionales llamadas PROGRAMADORES y ALTAMENTE_PAGADOS, donde cada una contuviera sólo los números de los empleados relevantes. Segundo (y mucho más importante), también es posible usar el mecanismo de **vistas** para lograr un efecto similar. Por ejemplo, podríamos definir PROGRAMADORES y ALTAMENTE_PAGADOS como vistas de la tabla base EMP:

```
CREATE VIEW PROGRAMADORES
    AS SELECT EMP#, NOMEMP, SALARIO, POSICIÓN
       FROM   EMP WHERE POSICIÓN =
       'Programados ;

CREATE VIEW ALTAMENTEPAGADOS
    AS SELECT EMP#, NOMEMP, SALARIO, POSICIÓN FROM
       EMP WHERE SALARIO > algún límite, digamos
       75000 ;
```

Y ahora, por supuesto, es perfectamente posible que el mismo "objeto" de empleado pertenezca simultáneamente a dos o más "colecciones". Además, la pertenencia en esas colecciones

—que son vistas— es manejada automáticamente por el sistema y no manualmente por el programador.

Finalizamos esta explicación mencionando un paralelismo esclarecedor entre los objetos mutables de los sistemas de objetos y las **variables dinámicas explícitas** de determinados lenguajes de programación (las variables BASED de PL/I son un caso de esto). Al igual que los objetos mutables de una clase dada, puede existir cualquier cantidad de variables dinámicas explícitas distintas para un tipo dado, y su almacenamiento es asignado en tiempo de ejecución mediante una acción explícita de programa. Además, esas variables distintas —otra vez al igual que los objetos mutables individuales— *no tienen nombre* y por lo tanto sólo pueden ser referidas mediante apuntadores. En PL/I, por ejemplo, podemos escribir:

```
DCL XYZ INTEGER BASED ; XYZ es una variable BASED    */ P es una variable
DCL P POINTER ;         de apuntador */
                        /* crea un nuevo ejemplar de XYZ /* y pone a P
ALLOCATE XYZ SET ( P )  para que apunte a él

P -> XYZ = 3 ;                      /* asigna el valor 3 al ejemplar de */
                                    /* XYZ apuntado por P               */
```

(y así sucesivamente). Este código PL/I se parece mucho al código de objeto que mostramos anteriormente; en particular, la declaración de la variable BASED está relacionada con la creación de una clase de objetos, y la operación ALLOCATE está relacionada con la creación de un NUEVO ejemplar de esa clase. Por lo tanto, podemos ver que la razón por la cual los OIDs son necesarios en el modelo de objetos es que precisamente los objetos que ellos identifican no poseen (en general) algún otro nombre único, al igual que los ejemplares de variables BASED en PL/I.

Jerarquías de clase

Ningún tratamiento de conceptos de objetos básicos estaría completo sin alguna mención a las **jerarquías de clase** (no se confunda con las jerarquías de contención). Sin embargo, el concepto de "jerarquía de clase" es esencialmente el mismo que el concepto de jerarquía de tipo que ya tratamos ampliamente en el capítulo 19; por lo tanto, nos contentamos aquí con algunas definiciones breves (tomadas en su mayoría del capítulo 19) y con algunas observaciones importantes. *Nota:* Le recordamos que hay poco consenso —en el mundo de los objetos o en cualquier otro— sobre un *modelo* de herencia abstracto y por lo tanto, los distintos sistemas de herencia difieren considerablemente entre sí en el nivel de detalle.

En primer lugar, decimos que la clase de objetos *Y* es una **subclase** de la clase de objetos *X* —en forma equivalente, decimos que la clase de objetos *X* es una **superclase** de la clase de objetos *Y*— si y sólo si todo objeto de la clase *Y* es necesariamente un objeto de la clase *X ("Y* ES UN *X").* Por lo tanto, los objetos de la clase *Y* **heredan** las variables de ejemplar públicas y los métodos que se aplican a la clase *X.* A la herencia de las variables de ejemplar se le conoce como herencia **estructural** y a la herencia de métodos se le conoce como herencia de **comportamiento.** Por supuesto, en un sistema puro sólo hay herencia de comportamiento y no herencia estructural —al menos para los objetos escalares o completamente encapsulados— debido a que no hay estructura a heredar (es decir, ninguna estructura visible para el usuario). Sin embargo, en la práctica los sistemas de objetos generalmente no son puros, y soportan algún grado de herencia estructural (lo que significa, para enfatizar, la herencia de las variables de ejemplar públicas).

Si la clase y es una subclase de la clase *X*, el usuario siempre puede usar un objeto y donde sea permitido un objeto *X* (por ejemplo, como argumento para diversos métodos) —éste es el principio de **sustituibilidad**— y por lo tanto, puede lograr la **reutilización del código.** Sin embargo, debido a que los sistemas de objetos a menudo no distinguen claramente entre valores y variables —es decir, entre objetos inmutables y mutables— tienden a caer en problemas sobre la distinción entre la sustituibilidad entre valores y las variables (vea el capítulo 19 para una mayor explicación). Sea como fuere, a la habilidad de aplicar el mismo método a objetos de la clase X y de la clase *Y*, se le conoce como polimorfismo (de inclusión).

El sistema vendrá equipado con determinadas jerarquías de clase integradas. En OPAL, por ejemplo (vea la sección 24.4), toda clase es considerada como una subclase, en cierto nivel, de la clase integrada OBJECT (debido a que "todo es un objeto"). Las subclases integradas de OBJECT incluyen BOOLEAN, CHAR, INTEGER, COLLECTION (etcétera); COLLECTION, a su vez, tiene una subclase llamada BAG, y BAG tiene otra llamada SET (etcétera).

Por último, algunos sistemas de objetos soportan cierta forma de herencia **múltiple** además de la herencia simple. Sin embargo, ningún sistema que yo conozca soporta la herencia de tupias o relaciones (ni simple ni múltiple) en el sentido de la referencia [3.3].

24.4 UN EJEMPLO DE INICIO A FIN

Ya hemos presentado los conceptos básicos de los sistemas de objetos. En esta sección mostraremos la manera en que son reunidos estos conceptos con un ejemplo de "inicio a fin"; es decir, mostraremos cómo puede ser definida una base de datos de objetos, cómo puede ser poblada y cómo podemos realizar las operaciones de recuperación y actualización a partir de ésta. Nuestro ejemplo está basado en el producto GemStone (de GemStone Systems Inc.) y su lenguaje de datos OPAL [24.14]; OPAL, a su vez, está basado en Smalltalk [24.26], uno de los lenguajes de objetos más puros (y ésta es la razón por la que lo usamos aquí). *Nota:* Debemos añadir que, al momento de la publicación de este libro, parece que Smalltalk ha sido desplazado en el mercado por C++ y —cada vez más— por Java, a pesar del hecho de que estos dos lenguajes son menos "puros" que Smalltalk.

El ejemplo involucra una versión simplificada de la base de datos de educación del ejercicio 8.10 del capítulo 8. La base de datos contiene información sobre un esquema de capacitación interno de una compañía. Para cada curso de capacitación, la base de datos contiene los detalles de todas las ofertas de ese curso; para cada oferta contiene detalles de todas las inscripciones de alumnos y de todos los maestros de esa oferta. La base de datos también contiene información acerca de los empleados. Una versión relaciona! de la base de datos se ve (en bosquejo) más o menos de esta forma:

```
CURSO       { CURSO#, TITULO }
OFERTA      { CURSO*, 0F#, FECHAOF, UBICACIÓN }
MAESTRO     { CURSO*, 0F#, EMP* }
INSCRIPCIÓN { CURSO*, 0F#, EMP*, CALIF }
EMP         { EMP*, NOMEMP, SALARIO, POSICIÓN }
```

La figura 24.6 es un diagrama referencial para esta base de datos. En caso de que requiera explicaciones adicionales, consulte los ejercicios y respuestas del capítulo 8.

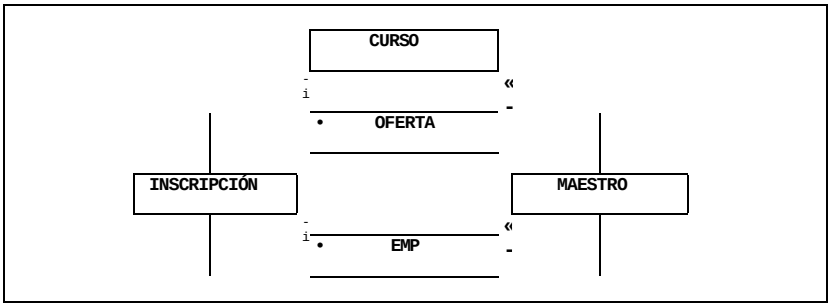


Figura 24.6 Diagrama referencia! para la base de datos de educación.

Definición de datos

Ahora procedemos a mostrar un conjunto de definiciones OPAL para esta base de datos. Aquí está primero la definición para una clase de objetos llamada EMP (las líneas están numeradas para efectos de referencias subsecuentes):

```
1  OBJECT SUBCLASS : 'EMP[1]
2     INSTVARNAMES : #[ ' EMP#' , ' NOMEMP', 'POSICIÓN' ]
3     CONSTRAINTS  : #[ #[ #EMP#, STRING ] ,
4                       [ #N0MEMP, STRING ] ,
5                       [ #P0SICION, STRING ] ] .
```

Explicación: La línea 1 define una clase de objetos llamada EMP, una subclase de la clase integrada llamada OBJECT. (En términos de OPAL, la línea 1 está *enviando un mensaje* al objeto OBJECT pidiéndole que llame al método SUBCLASS; INSTVARNAMES y CONSTRAINTS especifican argumentos para esa llamada al método. La definición de una clase nueva —al igual que cualquier otra cosa en OPAL— se realiza por lo tanto enviando un mensaje hacia un objeto.) La línea 2 establece que los objetos de la clase EMP tienen tres variables de ejemplar privadas llamadas EMP#, NOMEMP y POSICIÓN, respectivamente. Y las líneas 3 a 5 restringen esas variables de ejemplar para que cada una contenga objetos de la clase STRING. *Nota: A* lo largo de esta sección omitimos la explicación de los detalles puramente sintácticos (como los signos "#" que aparecen por todos lados) que son esencialmente irrelevantes para nuestro propósito.

Para repetir, las variables de ejemplar EMP#, NOMEMP y POSICIÓN son *privadas* de la clase EMP; por lo tanto, pueden ser accedidas por nombre únicamente dentro del código que implementa métodos para esa clase. Aquí, por ejemplo, están las definiciones de los métodos para "obtener y colocar" —es decir, recuperar y actualizar— los números de empleado (el símbolo ^ puede ser leído como "regresar"):

```
METHOD  EMP
  OBTENER_EMP#
%          ~EMP#

METHOD  EMP

  PONEF  1 EMP# : EMP  PARM
         EMP# := EMP  PARM
```

En la siguiente subsección tendremos más que decir con respecto a la definición de métodos. Mientras tanto, ésta es la definición de la clase CURSO:

```
1 OBJECT SUBCLASS   'CURSO'
2   INSTVARNAMES    #[   ' CURSO//' , ' TITULO1, 'OFERTAS
3   CONSTRAINTS     #[           #[ STRING ] ,
4                         [ //TTTIII∩  STRING ) ,
5                         [ #OFERTAS , OSET 1 1 .
```

Explicación: La línea 5 especifica que la variable de ejemplar privada OFERTAS contendrá el **OÍD** de un objeto de la clase OSET (una clase que definiremos dentro de unos momentos). De manera informal, OFERTAS indicará el conjunto de todas las ofertas para el curso en cuestión; en otras palabras, hemos decidido modelar el vínculo curso-ofertas como una jerarquía de contención, en la cual las ofertas están contenidas conceptualmente dentro del curso correspondiente.

Luego la clase OFERTA:

```
1  OBJECT SUBCLASS : 'OFERTA'
2     INSTVARNAMES : #[ '0F#', 'FECHAO', 'UBICACIÓN',
3                                 INSCRIPCIONES', 'MAESTROS' ]
4     CONSTRAINTS  : #[ #[ #0F#, STRING ] ,
5                          [ #FECHA0, DATETIME ] ,
6                          [ #UBICACION, STRING ] ,
7                          [ //INSCRIPCIONES, NSET ] ,
8                          [ #MAESTROS, TSET J J .
```

Explicación: La línea 7 especifica que la variable de ejemplar privada INSCRIPCIONES contendrá el OÍD de un objeto de la clase NSET; de manera informal, INSCRIPCIONES indicará el conjunto de todas las inscripciones para la oferta en cuestión. De manera similar, MAESTROS indicará el conjunto de todos los maestros para la oferta en cuestión. Por lo tanto, de nuevo estamos adoptando una representación de jerarquía de contención. Vea posteriormente las definiciones de NSET y TSET.

Luego la clase INSCRIPCIÓN:

```
1  OBJECT SUBCLASS  : 'INSCRIPCIÓN'
2     INSTVARNAMES  : #[ 'EMP', 'CALIF' ]
3     CONSTRAINTS   : #[ #[ #EMP, EMP ] ,
4                          [ #CALIF, STRING ] ]
```

Explicación: La variable de ejemplar privada EMP (línea 3) contendrá el OÍD de un objeto de la clase EMP, para que represente al empleado individual al que esta inscripción hace referencia. *Nota:* Hemos colocado el objeto EMP "dentro" del objeto INSCRIPCIÓN correspondiente para continuar con la representación de jerarquía de contención. Pero observe la asimetría: las inscripciones son un vínculo muchos a muchos, pero los participantes en ese vínculo —empleados y ofertas— están siendo tratados en forma diferente.

Por último, los maestros. Para efectos del ejemplo nos separamos ligeramente de la versión relacional original de la base de datos y tratamos a los maestros como una subclase de los empleados:

```
1  EMP SUBCLASS : 'MAESTRO'
2       INSTVARNAMES : #[ 'CURSOS' ]
3       CONSTRAINTS  : #[ #[ #CURSOS, CSET ] ] .
```

Explicación: La línea 1 define una clase de objetos llamada MAESTRO, una subclase de la clase EMP definida por el usuario (en otras palabras, MAESTRO "ES UN" EMP). Por lo tanto,

cada objeto individual MAESTRO tiene variables de ejemplar privadas EMP#, NOMEMP y POSICIÓN (todas heredadas de EMP*), además de CURSOS que contendrá el OÍD de un objeto de la clase CSET; ese objeto CSET indicará el conjunto de todos los cursos que este maestro puede enseñar. Cada objeto MAESTRO también hereda todos los métodos de EMP.

Como ya indicamos, las definiciones de clase anteriores suponen la existencia de varias clases de colección (ESET, CSET, OSET, NSET y TSET). Aquí están las definiciones de estas clases:

```
1  SET SUBCLASS   : 'ESET'
2       CONSTRAINTS :  EMP .
```

Explicación: La línea 1 define una clase de objetos ESET, una subclase de la clase SET integrada. La línea 2 restringe a los objetos de la clase ESET para que sean conjuntos de OIDs de la clase EMP. En general, podría existir cualquier cantidad de objetos de la clase ESET, pero crearemos solamente uno (vea la siguiente subsección) que será el conjunto de los OIDs de *todos* los objetos EMP que existen actualmente en la base de datos. De manera informal, ese único objeto ESET puede ser considerado como el equivalente en objetos de la varrel base EMP para la versión relacional de la base de datos.

Las definiciones de CSET, OSET, NSET y TSET son similares (vea más adelante). Sin embargo, en cada uno de estos casos tendremos en definitiva que crear varios objetos de la clase de colección relevante y no sólo uno; por ejemplo, existirá un objeto de colección OSET independiente para cada objeto CURSO individual.

```
S  SUBCLASS     | 'CSET'[1]
F  CONSTRAINTS  | CURSO .
S  SUBCLASS     | 'OSET'[1]
E  CONSTRAINTS  | OFERTA .
S  SUBCLASS     | 'NSET'[1]
E  CONSTRAINTS  | INSCRIPCIÓN
S  SUBCLASS     | 'TSET'
E  CONSTRAINTS  | MAESTRO .
```

Población de la base de datos

Ahora consideremos lo que implica poblar la base de datos. Consideremos cada una de las cinco clases de objetos básicas (EMP, CURSO, etcétera) a la vez. Primero los empleados. Recuerde que pretendemos recolectar los OIDs en conjunto de todos los objetos EMP que existen actualmente en un objeto ESET, y por lo tanto, primero necesitamos crear ese objeto ESET:

```
OID_DE_SET_DE_TODOS_EMPS  :=  ESET NEW .
```

La expresión que está al lado derecho de esta asignación regresa el OÍD de un nuevo ejemplar vacío de la clase ESET (es decir, un conjunto vacío de OIDs de EMP); el OÍD de ese nuevo ejemplar es asignado entonces a la variable de programa OID_DE_SET_DE_TODOS_EMPS. De manera *muy* informal, diremos que OID_DE_SET_DE_TODOS_EMPS indica el "conjunto de todos los empleados".

---

[1] Observe que ésta es la representación *privada* (es decir, la implementación física) que está siendo heredada aquí.

Ahora, cada vez que creamos un nuevo objeto EMP queremos que el OÍD de ese objeto sea insertado en el objeto ESET identificado por el OÍD que acaba de ser guardado en la variable OID_DE_SET_DE_TODOS_EMPS. Por lo tanto, definimos un *método* para la creación de dicho objeto EMP e insertamos su OÍD en ese objeto ESET. (En forma alterna, podríamos escribir un programa de aplicación para que realizara la misma tarea.) Éste es el método:

```
 1  METHOD : ESET                                          " ¡anónimo!
 2   AGREGAR_EMP#    : EMP# _PARM                          " parámetros
 3   AGREGARNOMEMP   : NOMEMP PARM
 4   AGREGAR_POS     : POS_ PARM
 5   I OID_EMP       I                                     " variable local
 6   OID_EMP := EMP  NEW .                                 " nuevo empleado
 7   OID_EMP PONER_  EMP#    : EMP#_PARM ;                 " inicia
 8           PONER_  NOMEMP  : NOMEMPPARM ;
 9           PONER   POS     : POS_PARM .
10   SELF ADD: OID_  EMP                                   " inserta
 1  %
 1
```

Explicación:

La línea 1 define el código que viene a continuación (hasta el signo terminal de porcentaje que está en la línea 11) para que sea un método aplicable a objetos de la clase ESET. (De hecho, es claro que en tiempo de ejecución sólo existirá *exactamente un* objeto de la clase ESET en el sistema.)

Las líneas 2 a 4 definen tres parámetros, con los nombres externos AGREGAR_EMP#, AGREGAR_NOMEMP y AGREGAR_POS. Estos nombres son usados en los mensajes que llaman al método. Los nombres internos correspondientes EMP#_PARM, NOMEMP_PARM y POS_PARM son usados en el código que implementa el método.

La línea 5 define que OID_EMP sea una variable local y la línea 6 asigna entonces a esa variable el OÍD de un ejemplar nuevo de EMP sin iniciar.

Las líneas 7 a 9 envían un mensaje a ese nuevo ejemplar de EMP; el mensaje especifica que se invoquen tres métodos (PONER_EMP#, PONER_NOMEMP y PONER_POS) y pasan un argumento para cada uno de ellos (EMP#_PARM para PONER_EMP#, NOMEMP_PARM para PONER_NOMEMP y POS_PARM para PONER_POS). *Nota:* Estamos suponiendo que los métodos PONER_NOMEMP y PONER_POS, similares al método PONER_EMP# que mostramos anteriormente, también han sido ya definidos.

La línea 10 envía un mensaje a SELF, que es un símbolo especial que indica el objeto al cual se le aplica actualmente —en tiempo de ejecución— el método que está siendo definido (es decir, el objeto de *destino* actual). El mensaje hace que se aplique el método integrado ADD a ese objeto (ADD es un método que es entendido por todas las colecciones); el efecto en este caso es insertar el OÍD del objeto identificado por OID_EMP dentro del objeto identificado por SELF (el cual será el objeto ESET que contiene los OIDs de todos los objetos EMP que existen actualmente). *Nota:* La razón por la que esta variable especial SELF es necesaria, es porque que el parámetro correspondiente al objeto de destino no tiene nombre por sí mismo (vea la línea 1).

Observe que —como señalamos en el comentario acerca de la línea 1— el método que está siendo definido tampoco tiene nombre. En general, los métodos de hecho no tienen nombre en OPAL, sino que en su lugar están identificados por su *signatura* (que está definida

en OPAL como la combinación del nombre de la clase a la cual se aplica y los nombres externos de sus parámetros). Esta convención puede conducir a ambigüedades molestas, como puede ver. Observe también otra implicación ligeramente desafortunada; es decir, que si dos métodos se aplican a la misma clase y toman los mismos parámetros, a esos parámetros les debemos dar (de manera arbitraria) nombres externos diferentes en los dos métodos.

Ahora tenemos un método para la inserción de EMPs nuevos en la base de datos, pero de hecho todavía no hemos insertado ninguno. Por lo tanto, hagámoslo:

```
0ID_DE_SET_DE_TOD0S_EMPS AGREGAR_EMP#  : 'E089'
                         AGREGARNOMEMP : 'Helms'
                         AGREGARPOS    : 'Conserje'
```

Esta instrucción crea un objeto EMP para el empleado E009 y agrega el OÍD de ese objeto EMP al conjunto de OIDs de todos los objetos EMP existentes actualmente.

A propósito, observe que el método NEW integrado nunca debe ser usado ahora sobre la clase EMP, sino sólo como parte del método que acabamos de definir. De no hacerlo así, podríamos crear algunos objetos EMP colgantes; es decir, empleados que no están representados en el objeto ESET que contiene a los OIDs de todos los objetos EMP que existen actualmente. *Nota:* Pedimos disculpas por la repetición insistente de ambigüedades molestas como "el método que acabamos de definir" y "el objeto ESET que contiene los OIDs de todos los objetos EMP que existen actualmente", pero es difícil hablar acerca de cosas que no tienen nombre.

Pasemos ahora a los cursos. Los empleados en realidad representan el caso más simple posible, ya que corresponden a "entidades normales" (para usar la terminología del modelo E/R) y además no contienen otros objetos incrustados dentro de ellos mismos (con excepción de los inmutables). Ahora pasamos a considerar el caso más complejo, los *cursos*, los cuales —aunque también son "entidades normales"— incluyen conceptualmente a otros objetos mutables incrustados dentro de ellos. A grandes rasgos, los pasos que debemos dar son los siguientes:

1. Aplicar el método NEW a la clase CSET para crear un "conjunto de todos los cursos" (en realidad, de OIDs de CURSO) inicialmente vacío.
2. Definir un método para crear un nuevo objeto CURSO y para agregar su OÍD al "conjunto de todos los cursos". Ese método tomará un CURSO# y un TITULO especificados como argumentos y creará un nuevo objeto CURSO con los valores especificados. También apli cará el método NEW a la clase OSET para crear un conjunto de OIDs de OFERTA inicialmente vacío, y luego colocará el OÍD de ese conjunto vacío de OIDs de oferta en la posición OFERTAS dentro del nuevo objeto CURSO.
3. Llamar al método que acaba de ser definido, para cada curso individual a la vez.

Luego pasamos a las ofertas. Esta vez los pasos son los siguientes:

1. Definir un método para crear un nuevo objeto OFERTA. Este método tomará un OF#, una FECHAO y una UBICACIÓN especificados como argumentos y creará un nuevo objeto OFERTA con esos valores especificados. También:
   - Aplicará el método NEW a la clase NSET para crear un conjunto de OIDs de INSCRIPCIÓN inicialmente vacío, y luego colocará el OÍD de ese conjunto vacío en la posición INSCRIPCIONES dentro del nuevo objeto OFERTA.

- Aplicará el método NEW a la clase TSET para crear un conjunto de OIDs de MAESTRO inicialmente vacío, y luego colocará el OÍD de ese conjunto vacío en la posición MAES TROS dentro del nuevo objeto OFERTA.

2. El método también tomará un valor CURSO# como argumento y usará ese valor CURSO# para:
   - Localizar el objeto CURSO correspondiente para el nuevo objeto OFERTA (vea la si guiente subsección para conocer la manera en que puede hacerlo). *Nota:* Por supuesto, si no puede encontrar el curso correspondiente, el método debe rechazar el intento de crear una nueva oferta. A lo largo de estas explicaciones omitimos las consideraciones detalladas de dichas excepciones.
   - Por lo tanto, localizar el "conjunto de todas las ofertas" para ese objeto CURSO.
   - Por lo tanto, agregar el OÍD del nuevo objeto OFERTA al "conjunto de todas las ofer tas" adecuado.

   Por lo tanto, observe cuidadosamente que (como mencionamos anteriormente en este capítulo) los OIDs no evitan la necesidad de claves de usuario tales como CURSO#. De hecho, dichas claves son necesarias no sólo para hacer referencia a objetos que están fuera de la base de datos, sino que también sirven como base para determinadas búsquedas dentro de ella.
3. Por último, llamar al método que justo acabamos de definir, para cada oferta individual a la vez.

A propósito, observe que (para mantenernos con nuestro diseño de jerarquía de contención) hemos decidido no crear "un conjunto de *todas* las ofertas". Una implicación de esta omisión es que cualquier consulta que requiera a ese conjunto como su alcance —por ejemplo, "obtener todas las ofertas en Nueva York"— involucrará una determinada cantidad de código procedural para obtenerlo (vea la siguiente subsección).

Luego, las inscripciones. La diferencia en tipo entre los casos de inscripciones y ofertas es que los objetos INSCRIPCIÓN incluyen una variable de ejemplar, EMP, cuyo valor es el OÍD del objeto EMP relevante. Por lo tanto, la secuencia de pasos necesaria es la siguiente:

1. Definir un método para crear un nuevo objeto INSCRIPCIÓN. Este método tomará un CURSO#, un OF#, un EMP# y una CALIF especificados como argumentos, y creará un nue vo objeto INSCRIPCIÓN con el valor de CALIF especificado. Luego:
   - Usará los valores de CURSO# y OF# para localizar el objeto OFERTA correspondiente para el nuevo objeto INSCRIPCIÓN.
   - Por lo tanto, localizará "el conjunto de todas las inscripciones" para ese objeto OFERTA.
   - Por lo tanto, añadirá el OÍD del nuevo objeto INSCRIPCIÓN al "conjunto de todas las inscripciones" adecuado.

   También:
   - Usará el valor EMP# para localizar el objeto EMP relevante.
   - Por lo tanto, colocará el OÍD de ese objeto EMP en la posición EMP dentro del nuevo objeto INSCRIPCIÓN.
2. Llamar al método que acabamos de definir, para cada inscripción individual a la vez.

Por último, los maestros. La diferencia en tipo entre los maestros y las ofertas es que hemos hecho que MAESTRO sea una subclase de EMP. Por lo tanto:

1. Definir un método para la creación de un nuevo objeto MAESTRO. Ese método tomará un CURSO#, un OF# y un EMP# especificados como sus argumentos. Luego:
   - Usará el valor EMP# para localizar el objeto EMP relevante.
   - Cambiará la clase más específica de ese objeto EMP a MAESTRO (ya que ese empleado ahora es además un maestro). La manera en que se realiza ese cambio de clase es alta mente dependiente del sistema específico que esté bajo consideración (vea el capítulo 19) y aquí omitimos los detalles específicos.

   También:

   - Usará los valores de CURSO# y OF# para localizar el objeto OFERTA correspondiente para el nuevo objeto MAESTRO.
   - Por lo tanto, localizará "el conjunto de todos los maestros" para ese objeto OFERTA.
   - Por lo tanto, agregará el OÍD del nuevo objeto MAESTRO al "conjunto de todos los maestros" adecuado.
2. Además, es necesario especificar de alguna forma el conjunto de cursos que este maestro puede enseñar, así como ajustar adecuadamente la variable de ejemplar CURSOS en el nuevo objeto MAESTRO. Aquí omitimos los detalles para este paso.
3. Llamar al método que acabamos de definir, para cada maestro individual a la vez.

Operaciones de recuperación

Antes de entrar en detalles sobre las operaciones de recuperación como tales, señalamos (aunque ya debe ser obvio) que OPAL —al igual que los lenguajes de objetos en general— es esencialmente un lenguaje de registros (o de objetos) y no un lenguaje de conjuntos. Por lo tanto, la mayoría de los problemas requerirán que algún programador escriba un código procedural. Consideraremos un solo ejemplo: la consulta "obtener todas las ofertas de Nueva York del curso C001". Suponemos, para efectos del ejemplo, que tenemos una variable llamada OOSOAC cuyo valor es el OÍD del "conjunto de todos los cursos". Éste es el código:

```
1  | CURSOC001 , C001OFS , C001NYOFS |
2   CURSO_C001
3       := OOSOAC DETECT : [ :CX | ( CX OBTENER_CURSO# ) = 'C001' ] .
4   C001OFS
5       := CURSOC001 OBTENEROFERTAS .
6   C001NYOFS
7       := C001OFS SELECT :
8                [ :0X | ( OX OBTENERUBICACION ) ■ 'Nueva York' ] .
9   " C001_NY_OFS .
```

Explicación:

- La línea 1 declara tres variables locales: CURSO_C001, que se usará para guardar el OÍD del curso C001; C001_OFS, que se usará para guardar el OÍD del conjunto de OIDs de las

ofertas del curso C001; y C001_NY_OFS, que se usará para guardar el OÍD del conjunto de los OIDs de las ofertas requeridas actualmente (es decir, las de Nueva York).

- Las líneas 2 y 3 envían un mensaje al objeto (de colección) indicado por la variable OOSOAC. El mensaje llama al método integrado DETECT para que sea aplicado a esa colección. El argumento de DETECT es una expresión de la forma:

  ```
  [ -X I P(X) )
  ```

  Aquí *p(x)* es una expresión condicional que involucra a la variable *x,* y *x* es en efecto una variable de alcance sobre los miembros de la colección a la cual se aplica DETECT (es decir, el conjunto de objetos CURSO del ejemplo). El resultado de DETECT es el OÍD del primer objeto *x* encontrado en ese conjunto, para el cual *p(x)* da como resultado *verdadero;* es decir, es el objeto CURSO para el curso C001, en el ejemplo.* Después, el OÍD de ese objeto CURSO es asignado a la variable CURSO_C001. *Nota:* También es posible especificar un argumento de "escape" para que DETECT maneje el caso en el cual *p(x)* nunca da como resultado *verdadero.* Aquí omitimos los detalles.
- Las líneas 4 y 5 asignan a la variable C001_OFS el OÍD del "conjunto de todas las ofertas" para el curso C001.
- Las líneas 6 a 8 son muy similares a las líneas 2 y 3. La operación del método integrado SELECT es la misma que la de DETECT, con excepción de que regresa el OÍD del con junto de OIDs de *todos* los objetos *x* (en lugar de solamente el primero) para los cuales *p(x)* da como resultado *verdadero.* Por lo tanto, en el ejemplo, el efecto es asignar a la variable C001_NY_OFS, el OÍD del conjunto de los OIDs de las ofertas de Nueva York del curso C001.
- Por último, la línea 9 regresa ese OÍD a quien lo llamó.

En este ejemplo surgen estos puntos:

1. Primero, observe que la expresión condicional *p(x)* en SELECT y DETECT puede involu crar (en su forma más compleja) un conjunto de comparaciones escalares simples unidas por AND; es decir, no es una condición de búsqueda arbitrariamente compleja.
2. Los corchetes que rodean a la expresión general de argumento en SELECT y DETECT, pueden ser reemplazados por llaves. Si utiliza llaves, OPAL tratará de usar un índice (en caso de que exista) en la aplicación del método. Si utiliza corchetes no lo hará.
3. Cuando decimos que DETECT regresa el OÍD del "primer" objeto encontrado que hace que *p(x)* dé como resultado *verdadero,* nos referimos por supuesto al primero, de acuerdo con cualquier secuencia que OPAL use para buscar en el conjunto (los conjuntos no tienen un ordenamiento intrínseco propio). En nuestro ejemplo esto no hace diferencia alguna, ya que el "primer" objeto que hace que la condición sea verdadera es, de hecho, el único.

---

*A lo largo de este ejemplo, damos por hecho que ya han sido definidos métodos como OBTENER_CURSO#, similares al método OBTENER_EMP# que mostramos anteriormente en esta sección.

4. Como probablemente ya haya observado, hemos estado haciendo un amplio uso de expre siones tales como "el método DETECT"; aunque, como señalamos anteriormente, en OPAL los métodos no tienen nombre. Además, DETECT y SELECT no son nombres de método (y las frases como "el método DETECT" son incorrectas en el sentido estricto). En su lugar, hay nombres de parámetros externos para determinados métodos integrados que no tienen nombre. Sin embargo, por razones de brevedad y simplicidad, continuaremos hablando como si DETECT y SELECT (y otros conceptos similares) fueran nombres de método.
5. Tal vez se dio cuenta que hemos estado usando (¡muchas veces!) la expresión "el método NEW". De hecho, este uso *no* es incorrecto. Los métodos que no toman argumentos, además de un destino, son una excepción a la regla general de OPAL de que los métodos no tienen nombres.

Operaciones de actualización

En la subsección anterior ya tratamos la analogía de objetos en la operación INSERT relacional. A continuación explicamos las analogías de objetos en UPDATE y DELETE.

- *Actualización.* Las operaciones de actualización se realizan esencialmente de la misma forma que las de recuperación, con excepción de que se usan los métodos PONER_ en vez de OBTENER..
- *Eliminación.* Se utiliza el método integrado REMOVE para eliminar objetos (para ser más precisos, se usa para eliminar el OÍD de un objeto especificado de una colección especifi cada). Cuando un objeto llega a un punto en el que ya no hay ninguna referencia a él —es decir, ya no puede ser accedido de ninguna forma— entonces OPAL lo elimina automáti camente por medio de un proceso de recolección de basura. Éste es un ejemplo:

```
E001  :=  0ID_DE_SET_DE_TODOS_EMPS
              DETECT : [ :EX I ( EX OBTENEREMP* ) ■ ' E001¹ ] .
OID_DE_SET_DE_TODOS_EMPS REMOVE : E001 .
```

("quita al empleado E001 del conjunto de todos los empleados").

Pero, ¿qué pasa si queremos hacer cumplir (digamos) una regla ON DELETE CAS-CADE?; por ejemplo, una regla para que la eliminación de un empleado también elimine todas las inscripciones de ese empleado. Por supuesto, la respuesta es que tenemos que implementar nuevamente un método adecuado; en otras palabras, tenemos que escribir algo más de código procedural.

A propósito, podría pensar que el enfoque de recolección de basura para la eliminación implementa al menos un tipo de regla ON DELETE RESTRICT, en tanto que un objeto no es eliminado realmente mientras existen referencias hacia ese objeto. Sin embargo, éste no es necesariamente el caso. Por ejemplo, los objetos OFERTA no incluyen el OÍD del objeto CURSO correspondiente y por lo tanto, las ofertas no "restringen" los DELETES sobre cursos. De hecho, las jerarquías de contención implican tácitamente un tipo de regla ON DELETE CASCADE, a menos que el usuario decida:

- Incluir el OÍD del padre en el hijo, o
- Incluir el OÍD del hijo dentro de algún otro objeto en cualquier lugar de la base de datos,

en cuyo caso, la interpretación de jerarquía de contención ya no tiene ningún sentido. Vea la explicación de *variables inversas* en la siguiente sección.

Por último, observe que puede usar REMOVE para emular la operación relacional DROP, por ejemplo, para desechar la clase INSCRIPCIÓN. Dejamos los detalles como ejercicio.

24.5 ASPECTOS VARIOS

En esta sección damos un breve vistazo a un grupo de temas algo mezclados, es decir:

- Consultas *ad hoc* y temas relacionados.
- Integridad.
- Vínculos.
- Lenguajes de programación de bases de datos.
- Consideraciones de rendimiento.
- ¿Un DBMS de objetos es en realidad un DBMS?

Consultas *ad hoc* y temas relacionados

De manera deliberada no enfatizamos el punto anterior, pero si realmente es cierto que los métodos predefinidos son la única forma para manipular objetos, entonces ¡las consultas *ad hoc* son imposibles!, a menos que las clases y métodos estén diseñados de acuerdo con una disciplina determinada muy específica. Por ejemplo, si los únicos métodos definidos para la clase DEPTO son (como sugerimos en la sección 24.2) CONTRATAR_EMP, DESPEDIR_EMP y RECORTAR_PRESUPUESTO, incluso una consulta tan simple como, "¿quién es el gerente del departamento de programación?" no podría ser manejada.

En esencia, por la misma razón en general también son imposibles las definiciones de vistas y restricciones de integridad declarativas sobre objetos; de nuevo, a menos que sigamos una determinada disciplina específica.

La solución que le recomendamos para estos problemas (es decir, la "disciplina específica" que tenemos en mente) es la siguiente:

1. Defina un conjunto de operadores ("operadores THE_") que exponga *alguna representación posible* para los objetos en cuestión, como explicamos en el capítulo 5.
2. Incruste adecuadamente los objetos en un marco relacional. En el siguiente capítulo, trata mos a detalle esta parte de la solución.

Sin embargo, por lo general los sistemas de objetos *no* siguen esta disciplina (no exactamente). En vez de ello:*

1. Por lo general, definen operadores que no exponen alguna representación *posible* sino, en su lugar, la representación *real* (vea la explicación sobre variables de ejemplar públicas

*De hecho, aquí suponemos que el sistema de objetos en cuestión soporta las consultas *ad hoc*, como sucede con la mayoría de los sistemas modernos. Sin embargo, con frecuencia los primeros sistemas no lo hacían, en parte por las razones que explicaremos posteriormente en esta sección.

en la sección 24.2). "Todos los productos DBMS de objetos requieren actualmente que [las variables de ejemplar] referidas en... consultas sean públicas" [24.38].

2. Por lo general, no soportan un marco relacional sino que, en su lugar, soportan una variedad de otros marcos basados en bolsas, arreglos, etcétera. Con respecto a esto último, le recordamos que sostenemos que las clases —es decir, los tipos— y las relaciones son necesarias y suficientes en el nivel lógico (vea el capítulo 3); por lo que se refiere al modelo núcleo, creemos de hecho que los arreglos y todo lo demás son innecesarios e indeseables. Conjeturamos que el énfasis sobre las colecciones que no sean relaciones (y de hecho el rechazo casi total a las relaciones) en los sistemas de objetos, es producto nuevamente de una confusión sobre los asuntos de modelo *contra* implementación.

Hay otro aspecto importante relacionado con las consultas *ad hoc:* ¿de qué clase es el resultado? Por ejemplo, supongamos que ejecutamos la consulta "obtener el nombre y el salario de los empleados del departamento de programación" contra la base de datos de departamentos y empleados de la sección 24.3. Presuntamente el resultado tiene variables de ejemplar (públicas) NOMEMP y SALARIO. Pero no hay ninguna clase en la base de datos que tenga esta estructura. Por lo tanto, ¿tendríamos que predefinir una clase de éstas antes de poder hacer la consulta? (observe las implicaciones de esto: ¡una clase con n variables de ejemplar requeriría al menos 2" de estas clases predefinidas para soportar sólo las operaciones de recuperación!) Y sin importar cuál fuera la clase del resultado, ¿qué métodos serían aplicados?

Preguntas similares se presentan en relación con las operaciones de junta. Si podemos juntar los objetos de departamento y empleado (de nuevo), ¿de qué clase es el resultado?, ¿qué métodos serían aplicados?

Posiblemente, debido a que tales preguntas son difíciles de responder en un marco de objetos puro, algunos sistemas de objetos soportan las operaciones de "trazado de ruta" [24.25], [24.47], en lugar de las juntas en sí. Si por ejemplo, tomamos en cuenta la base de datos OPAL de la sección 24.4, la siguiente podría ser una expresión de ruta válida:

```
INSCRIPCIÓN . OFERTA . CURSO
```

Con el significado: "acceder al único objeto CURSO, referido por el único objeto OFERTA, referido por este objeto INSCRIPCIÓN".* Por lo general, una analogía relacional de esta expresión involucraría dos juntas y una proyección. En otras palabras, trazar la ruta implica el acceso a lo largo solamente de rutas predefinidas (de hecho, al igual que en los sistemas prerrelacionales) hacia objetos sólo de clases predefinidas (de nuevo, al igual que en los sistemas prerrelacionales).

Integridad

En el capítulo 8 dijimos que la integridad de los datos es absolutamente fundamental. Sin embargo, los sistemas de objetos —incluso aquellos que soportan consultas *ad hoc*— no soportan generalmente restricciones de integridad declarativas; en vez de ello, requieren que tales restricciones se hagan cumplir por medio de código procedural (es decir, por métodos o posiblemente por pro-

---

*De hecho, este ejemplo *no* es una expresión de ruta válida para la base de datos tal como la hemos definido, ¡ya que los apuntadores señalan hacia el lado erróneo! Por ejemplo, OFERTAS no hace referencia a CURSOS, sino que es referida por éstos.

gramas de aplicación). Por ejemplo, considere la siguiente restricción (o "regla de negocios") de la sección 8.5: "ningún proveedor con status menor que 20 puede proporcionar una parte en cantidad mayor a 500". El código procedural para hacer cumplir esta restricción tendrá que estar incluido, por lo general, al menos en todos los siguientes métodos:

- Método para crear un envío.
- Método para cambiar la cantidad del envío.
- Método para cambiar el status del proveedor.
- Método para asignar un envío a un proveedor diferente.

Surgen estos puntos y preguntas:

1. Hemos perdido obviamente la posibilidad de que el sistema determine por sí mismo cuándo hay que hacer la verificación de la integridad.
2. ¿Cómo nos aseguramos de que todos los métodos necesarios incluyan todo el código de cumplimiento necesario?
3. ¿Cómo impedimos que (por ejemplo) el usuario haga a un lado el método "crear envío" y use directamente el método integrado NEW sobre la clase de objetos de envío, con lo que también haría a un lado la verificación de la integridad?
4. Si cambian las restricciones, ¿cómo encontraremos todos los métodos que necesitan ser reescritos?
5. ¿Cómo nos aseguramos que el código de cumplimiento sea correcto?
6. ¿Cómo diferimos la verificación de la integridad (tiempo de COMMIT)?
7. ¿Qué hay acerca de las restricciones de transición?
8. ¿Cómo consultamos al sistema para que encuentre todas las restricciones aplicadas a un ob jeto dado o a una combinación de objetos?
9. ¿Se harán cumplir las restricciones durante la carga y el procesamiento de otras herramientas?
10. ¿Qué hay acerca de la optimización semántica (es decir, el uso de restricciones de integri dad para simplificar consultas, tal como explicamos en el capítulo 17)?

¿Y cuáles son las implicaciones de todo lo anterior sobre la productividad, tanto en la creación de la aplicación como en el mantenimiento subsiguiente?

Vínculos

Por lo general, los productos y la literatura relacionados con los objetos usan el término "vínculos" para referirse específicamente a aquellos vínculos que serían representados por las *claves extemas* en un sistema relacional. Y por lo general, proporcionan soporte de caso especial para este tipo especial de restricción de integridad, como lo mostramos a continuación. Considere nuevamente los departamentos y empleados. En un sistema relacional, los empleados tendrían normalmente una clave externa que hiciera referencia a los departamentos, y eso sería todo. Por el contrario, en un sistema de objetos existen al menos estas tres posibilidades:

1. Los empleados pueden incluir un OÍD que haga referencia a los departamentos. Esta posibilidad es similar al enfoque relacional aunque *no* idéntica (los OIDs y las claves externas no son lo mismo).

2. Los departamentos pueden incluir un conjunto de OIDs que hagan referencia a los em pleados. Esta posibilidad corresponde al enfoque de jerarquía de contención, tal como lo describimos en la sección 24.3.
3. Los enfoques 2 y 3 pueden ser combinados como indicamos aquí:

```
CLASS EMP ...
    ( ... DEPTO REF ( DEPTO ) INVERSE DEPTO.EMPS ) ... ; CLASS
DEPTO ...
    ( ... EMPS
          REF ( SET ( REF ( EMP ) ) ) INVERSE EMP. DEPTO ) ... ;
```

Observe la especificación INVERSE en las variables de ejemplar EMP.DEPTO y DEPTO.EMPS. Decimos que estas dos variables de ejemplar son inversas entre sí, ya que EMP.DEPTO es una variable de ejemplar de *referencia* y DEPTO.EMPS es una variable de ejemplar de *conjunto de referencia.* (Si ambas fueran variables de conjunto de referencia, el vínculo sería de muchos a muchos, en lugar de uno a muchos.)

Por supuesto, cada una de estas posibilidades requiere algún tipo de soporte de integridad referencial (vea más adelante). Sin embargo, hay primero una pregunta obvia que debemos hacer: ¿cómo manejan los sistemas de objetos los vínculos que involucran a más de dos clases, digamos proveedores, partes y proyectos? La mejor respuesta (es decir, la más simétrica) a esta pregunta parece requerir la creación de una clase "VPY" tal que cada objeto VPY individual tenga vínculos de "variables inversas" con el proveedor adecuado, la parte adecuada y el proyecto adecuado. Pero entonces —si tomamos en cuenta que la creación de una nueva clase de objetos es, aparentemente, el mejor enfoque para los vínculos de grado mayor que 2— se presenta la pregunta obvia de por qué los vínculos de grado 2 no son tratados de la misma forma.

También, con respecto a las variables inversas específicamente, ¿por qué es necesario introducir la asimetría, la direccionalidad y dos nombres diferentes para lo que en esencia es una cosa? Por ejemplo, los equivalentes de objetos de las dos expresiones relaciónales

```
VP.P# WHERE VP.V# = V#('V1')
VP.V# WHERE VP.P# = P#('pr)
```

se ven así:

```
V.PARTS.P# WHERE V.V# ■ V#('V1')
P.PR0VS.V# WHERE P.P# = P#('PV)
```

(sintaxis hipotética, elegida deliberadamente para resaltar las diferencias esenciales —por ejemplo, el uso de dos nombres de vínculo distintos, PARTS y PROVS— y evitar irrelevancias).

Integridad referencial. Como prometimos, ahora daremos una mirada más a fondo al soporte de objetos para la integridad referencial (la cual, dicho sea de paso, es frecuentemente aclamada como una fortaleza de los sistemas de objetos). Son posibles varios niveles de soporte. La siguiente taxonomía está tomada de Cattell [24.11]:

- *Ningún soporte del sistema.* La integridad referencial es responsabilidad del código escrito por el usuario (como era en el estándar de SQL original, dicho sea de paso).
- *Validación de referencia.* El sistema verifica que todas las referencias sean hacia objetos existentes del tipo correcto, aunque tal vez no permita la eliminación de objetos; en su lugar,

es posible que los objetos sean recolectados como basura cuando ya no queden referencias hacia ellos (como sucede en OPAL). Como explicamos en la sección 24.4, este nivel de soporte es aproximadamente equivalente (sólo aproximadamente) a (a) una regla ON DELETE CASCADE para subobjetos no compartidos dentro de una jerarquía de contención y (b) un tipo de regla ON DELETE RESTRICT para otros objetos (aunque sólo si "los apuntadores apuntan en la forma correcta").

Mantenimiento del sistema. Aquí el sistema mantiene actualizadas todas las referencias de forma automática (por ejemplo, estableciendo las referencias hacia objetos eliminados como *nil).* Este nivel de soporte es algo similar a la regla ON DELETE SET NULL.

"Semántica personalizada". ON DELETE CASCADE (fuera de una jerarquía de contención) podría ser un ejemplo de este caso. Al momento de la publicación de este libro, dichas posibilidades generalmente no son soportadas por los sistemas de objetos, sino que deben ser manejadas mediante código escrito por el usuario.

Lenguajes de programación de bases de datos

Los ejemplos de OPAL en la sección 24.4 ilustran que los sistemas de objetos no emplean, por lo general, el enfoque de "sublenguaje de datos incrustado" como lo hace SQL. En su lugar utilizan el mismo **lenguaje integrado** para las operaciones de bases de datos y para las que no lo son. Para usar la terminología del capítulo 2, el lenguaje anfitrión y el lenguaje de base de datos están *fuertemente acoplados* en los sistemas de objetos (de hecho, son lo mismo).

Ahora, es innegable que este enfoque tiene ventajas.* Una ventaja importante es que permite una mejor verificación de tipo [24.2]. Y la referencia [24.47] argumenta que la siguiente es otra:

> Con un solo lenguaje unificado no hay *desacoplo de impedancia* entre un lenguaje de programación procedural y un DML incrustado con semántica declarativa.

Aquí el término **desacoplo de impedancia** se refiere a la diferencia entre el nivel de registro simultáneo de los lenguajes de programación actuales y el nivel de conjunto simultáneo de los lenguajes de bases de datos como SQL. Y es cierto que esta diferencia de nivel hace que surjan problemas prácticos en los productos SQL. Sin embargo, ¡la solución a estos problemas *no* es llevar el lenguaje de base de datos al nivel de registro (que es lo que hacen los sistemas de objetos)! sino elevar el lenguaje de programación al nivel de conjunto. El hecho de que los lenguajes de objetos estén en el nivel de registro (es decir, de procedimientos) es una regresión a los días de los sistemas prerrelacionales como IMS e IDMS.

Para continuar con este último punto, también resulta que la mayoría de los lenguajes de objetos son por naturaleza de procedimientos, o "3GL". Por consecuencia, se pierden todas las ventajas relaciónales en el nivel de conjunto; en particular, se deteriora severamente la habilidad del sistema para optimizar las solicitudes del usuario, lo que significa que —al igual que en los sistemas prerrelacionales— el rendimiento queda en gran parte en manos de los usuarios (los programadores de aplicaciones o el DBA). Vea la subsección que viene a continuación.

---

*De hecho, el lenguaje de **Tutorial D** que usamos en gran parte del libro, adopta el mismo enfoque por razones que se describen en la referencia [3.3].

Consideraciones de rendimiento

El rendimiento bruto es uno de los principales objetivos para los sistemas de objetos. Para citar nuevamente a Cattell [24.11]: "una diferencia en rendimiento en un orden de magnitud puede en efecto constituir una diferencia *funcional,* ya que no es posible usar el sistema cuando el rendimiento está muy por debajo de los requerimientos" (parafraseado ligeramente). Muchos factores son importantes para el rendimiento. Estos son algunos de ellos:*

- *Agrupamiento.* Como vimos en el capítulo 17, el agrupamiento físico de los datos rela cionados lógicamente en el disco, es uno de los factores de rendimiento más importantes de todos. En general, los sistemas de objetos utilizan información lógica de la definición de base de datos (con respecto a las jerarquías de clase, jerarquías de contención u otros víncu los entre objetos declarados explícitamente) como una pista sobre la manera en que los datos deben estar agrupados físicamente. En forma alterna (y de preferencia), se le debe dar al DBA un control más directo y explícito sobre la transformación conceptual/interna (para utilizar nuevamente la terminología del capítulo 2).
- *Uso de memoria caché.* Los sistemas de objetos están orientados generalmente para ser usa dos en ambientes cliente-servidor. La recuperación desde el sitio servidor de los datos agru pados como una unidad (y por lo tanto —de manera ideal— relacionados lógicamente), y su almacenamiento en caché durante un amplio periodo en el sitio cliente, obviamente tiene sentido en tales ambientes.
- *Estafa.* El término "estafa" se refiere al proceso de reemplazar los apuntadores estilo OÍD —que por lo general son direcciones lógicas de disco— por direcciones de memoria prin cipal cuando los objetos son leídos en la memoria (o al revés, cuando los objetos son reescritos en la base de datos). Las ventajas para las aplicaciones que procesan "objetos complejos" y que requieren por lo tanto el manejo extenso de apuntadores, son obvias.
- *Ejecución de métodos en el servidor.* Considere la consulta "obtener todos los libros que tienen más de 20 capítulos". En un sistema relacional tradicional, los libros podrían estar representados como BLOBs[f] y las aplicaciones cliente tendrían que recuperar cada libro uno por uno y revisarlo para ver si tiene más de 20 capítulos. En un sistema de objetos, por el contrario, se ejecutaría el operador "obtener la cuenta de capítulos" en el servidor y se transmitirán al cliente únicamente los libros que en realidad se quieren.*

  Nota: En realidad, lo anterior no es un argumento a favor de los objetos, sino a favor de los *procedimientos almacenados* (vea los capítulos 8 y 20). Un sistema SQL tradicional con procedimientos almacenados dará aquí los mismos beneficios de rendimiento que un sistema de objetos con métodos.

---

*Además de los factores listados, podríamos argumentar que los sistemas de objetos logran un mejor rendimiento (en la medida en que lo hacen actualmente) "moviendo a los usuarios más cerca de la máquina"; es decir, exponiendo características (como los apuntadores) que deberían estar ocultas en la implementación.

[f] *Una nota con respecto a los BLOBs.* Aunque este tipo de datos no está incluido en el estándar de SQL/92, los productos SQL los han proporcionado tradicionalmente como una base para el manejo de "objetos bi-narios grandes" (aquí "objetos" significa objetos en sentido genérico y no en el sentido especial de los sis-temas de objetos). Un BLOB es, en esencia, sólo una cadena de bytes arbitrariamente larga; el sistema proporciona soporte de almacenamiento y recuperación para tales cadenas, pero nada más. Físicamente, los BLOBs a menudo están almacenados en un área especial propia de ellos, independiente del área de alma-cenamiento principal de las relaciones que los contienen de manera lógica (pueden ocupar muchas páginas físicas en el disco, y a menudo lo hacen).

*De hecho es una simplificación excesiva; los métodos con muchos datos se ejecutan mejor en el servidor, pero otros —por ejemplo, los que tienen muchos desplegados— tal vez se ejecuten mejor en el cliente.

La referencia [24.13] trata una prueba llamada OO1 para la medición del rendimiento del sistema en una base de datos de lista de materiales. La prueba involucra:

1. La recuperación aleatoria de 1,000 partes, aplicando en cada una un método definido por el usuario.
2. La inserción aleatoria de 1,000 partes, conectando cada una con otras tres.
3. La explosión aleatoria de partes (hasta siete niveles), aplicando un método definido por el usuario en cada una de las partes encontradas, junto con la implosión correspondiente.

De acuerdo con la referencia [24.13], la comparación de un producto de objetos (no especificado) con un producto SQL (no especificado) mostró una diferencia en rendimiento de hasta dos órdenes de magnitud a favor del sistema de objetos, en especial en accesos "calientes" (después de haber poblado la caché). Sin embargo, la referencia [24.13] también tiene el cuidado de decir que:

> Las diferencias... *no* deben ser atribuidas a una diferencia entre los modelos relacional y de objetos... Hay razones para creer que la mayoría de las diferencias pueden ser atribuidas a [cuestiones de implementación].

Esta queja es apoyada por el hecho de que las diferencias fueron mucho más pequeñas cuando la base de datos fue "grande" (es decir, cuando ya no era posible acomodar la base de datos completa en la caché).

Una prueba similar pero más amplia, llamada OO7, es descrita en la referencia [24.10].

¿Un DBMS de objetos es en realidad un DBMS?

Nota: Los comentarios de esta subsección fueron tomados en su mayoría de la referencia [24.18], la cual argumenta entre otras cosas, que los sistemas de objetos y los relaciónales son más diferentes de lo que uno se da cuenta. Para citar: "Las bases de datos de objetos surgieron por la necesidad que tenían los programadores de aplicaciones orientadas a objetos —por una variedad de razones específicas de las aplicaciones— de conservar sus objetos (de la aplicación) en una memoria persistente. Esa memoria persistente podría tal vez ser considerada como una base de datos, pero el punto importante es que *en realidad fue específica de la aplicación;* no era una base de datos compartida, de propósito general, que pretendiera ser adecuada para aplicaciones que no hubieran sido previstas al momento de definir la base de datos. Por consecuencia, muchas características que los profesionales de bases de datos ven como esenciales, sencillamente no fueron requerimientos en el mundo de los objetos, al menos no originalmente. Por lo tanto, hubo muy poca necesidad de:

1. Compartir datos entre aplicaciones.
2. Independencia física de los datos.
3. Consultas *ad hoc*.
4. Vistas e independencia lógica de los datos.
5. Restricciones de integridad declarativas, independientes de la aplicación.
6. Propiedad de los datos y mecanismos de seguridad flexibles.
7. Control de concurrencia.
8. Un catálogo de propósito general.
9. Un diseño de base de datos independiente de la aplicación.

"Estos requerimientos aparecieron posteriormente, después de que la idea básica de guardar objetos en una base de datos fue concebida por primera vez, y por lo tanto, todos constituyen características adicionales para el modelo original de objetos... Una consecuencia importante... es que en realidad hay una **diferencia de tipo** entre un DBMS de objetos y un DBMS relacional. De hecho, podríamos argumentar que un DBMS de objetos en realidad no es un DBMS, al menos no en el mismo sentido que un DBMS relacional es un DBMS. Por considerar que:

- Un DBMS relacional ya viene *listo* para ser usado. En otras palabras, tan pronto como el sistema está instalado, los usuarios... pueden comenzar a construir bases de datos, escribir aplicaciones, ejecutar consultas, etcétera.
- Un DBMS de objetos, por el contrario, puede ser visto como un *kit de construcción de DBMS.* Cuando es instalado por primera vez, *no* está disponible para su uso inmediato... En su lugar, primero debe ser *adaptado* por técnicos capaces y experimentados; dichos técnicos son quienes deben definir las clases y métodos necesarios, entre otras cosas (el sistema proporcio na un conjunto de bloques de construcción —herramientas de mantenimiento para bibliotecas de clases, compiladores de métodos, etcétera— para este propósito). Sólo cuando esta activi dad de adaptación haya terminado, el sistema estará disponible para que los programadores de aplicaciones y los usuarios finales lo utilicen; en otras palabras, el resultado de esa adaptación hará que se parezca más a un DBMS en el sentido más familiar del término.

"Observe, además, que el DBMS 'adaptado' resultante será *específico de una aplicación;* podría, por ejemplo, ser adaptado para aplicaciones CAD/CAM, pero ser esencialmente inútil para, por ejemplo, aplicaciones médicas. En otras palabras, todavía no será un DBMS de propósito general, en el mismo sentido que un DBMS relacional es un DBMS de propósito general".

El mismo artículo [24.18] también argumenta en contra de la idea —a la que llama frecuentemente **"persistencia ortogonal de tipos"** [24.2]— de acuerdo con la cual se permite que la base de datos incluya objetos (mutables) de una complejidad cualquiera: "el modelo de objetos requiere soporte para [una gran cantidad de] *generadores de tipo*... Los ejemplos incluyen STRUCT (o TUPLE), LIST, ARRAY, SET, BAG, etcétera..." Junto con los IDs de objetos, la disponibilidad de estos generadores de tipo significa esencialmente que *cualquier estructura de datos que puede ser creada en un programa de aplicación, puede ser creada como un objeto en una base de datos de objetos,* y además, que la estructura de dichos objetos es *visible ante el usuario.* Por ejemplo, considere el objeto, digamos EX, que es (o mejor dicho, indica) la colección de empleados en un departamento dado. Entonces, EX puede ser implementado como una lista enlazada o como un arreglo, y los usuarios tendrán que saber de cuál se trata (debido a que, por consiguiente, los operadores de acceso difieren).

Este enfoque de *todo se vale* con respecto a lo que puede ser almacenado en la base de datos, es un punto principal de diferencia entre los modelos de objetos y el relacional —por supuesto— y merece que lo expliquemos más. En esencia:

- El modelo de objetos dice que podemos almacenar cualquier cosa que queramos, cualquier estructura de datos que podamos crear con los mecanismos del lenguaje de programación usual.
- El modelo relacional efectivamente dice lo mismo, pero además insiste en que cualquier cosa que almacenemos debe ser presentada ante el usuario en forma relacional pura.

"Más precisamente, el modelo relacional —de manera adecuada— no dice *nada* acerca de lo que puede ser almacenado físicamente... Por lo tanto, no impone límites sobre cuáles estructuras

de datos son permitidas en el nivel físico; el único requerimiento es que cualquier estructura que de hecho *esté* guardada físicamente, debe ser transformada a relaciones en el nivel lógico y por lo tanto, debe estar oculta ante el usuario. Por lo tanto, los sistemas relaciónales hacen una distinción clara entre lo lógico y lo físico (el modelo de datos contra la implementación), mientras que los sistemas de objetos no la hacen. Una consecuencia es que —contrario a lo que normalmente se piensa— los sistemas de objetos pueden proporcionar perfectamente menos independencia de datos que los sistemas relaciónales. Por ejemplo, suponga que la implementación en alguna base de datos de objetos del objeto EX que mencionamos anteriormente (que indica la colección de empleados en un departamento dado) es cambiada de un arreglo a una lista enlazada. ¿Cuáles son las implicaciones para el código existente que tiene acceso a ese objeto EX? ".

24.6 RESUMEN

A manera de resumen, ésta es una lista de las características principales del modelo de objetos tal como lo hemos presentado, junto con una valoración subjetiva de cuáles características son esenciales, cuáles sería "bueno tener" pero sin que sean esenciales, cuáles es "malo tener", cuáles en realidad son ortogonales para la pregunta de si el sistema es un sistema de objetos o de algún otro tipo, etcétera. Este análisis allana el camino para nuestra explicación sobre los sistemas *objeto/relaciónales* en el capítulo 25.

- **Clases de objetos** (es decir, *tipos).* Obviamente son esenciales (en realidad, son la cons trucción más fundamental de todas).
- **Objetos.** Los propios objetos —tanto "mutables" como "inmutables"— son claramente esen ciales, aunque preferiríamos llamarlos simplemente *variables* y *valores,* respectivamente.
- **IDs de objetos.** Son innecesarios y de hecho también son indeseables (es decir, en el nivel del modelo), debido a que en esencia sólo son apuntadores. Vea la referencia [24.19] para una amplia explicación de este tema.
- **Encapsulation.** Como explicamos en la sección 24.2, "encapsulado" significa simple mente *escalar,* y preferiríamos usar ese término (recordando siempre que algunos "objetos" no son escalares de ninguna forma).
- **Variables de ejemplar.** Primero, las variables de ejemplar *privadas* (y también *protegidas)* son, por definición, simples cuestiones de implementación y por lo tanto no son relevantes para la definición de un modelo abstracto, que es lo que nos preocupa aquí. Segundo, las variables de ejemplar *públicas* no existen en un sistema de objetos puro y por lo tanto, tam poco son relevantes. Concluimos que las variables de ejemplar pueden ser ignoradas; los "objetos" deberán ser manipulados únicamente mediante "métodos".
- **Jerarquía de contención.** En la sección 24.3 explicamos que, en nuestra opinión, las jerar quías de contención son engañosas y en realidad tienen un nombre erróneo, ya que general mente contienen OIDs y no "objetos". *Nota:* Una jerarquía (no encapsulada) que realmente incluyera objetos en sí, sería permisible, aunque por lo general está contraindicada; sería similar, en cierta forma, a una varrel con atributos con valor de relación (vea las partes II y III de este libro).

- **Métodos.** El concepto por supuesto es esencial, aunque preferiríamos usar el término más convencional de *operadores.* Sin embargo, la integración de métodos con las clases *no* es esencial y conduce a varios problemas [3.3]; preferiríamos definir por separado a las "cla ses" (tipos) y a los "métodos" (operadores), como lo hicimos en el capítulo 5, y evitar por lo tanto la noción de "objetos destino".

  Existen determinados operadores sobre los que también quisiéramos insistir: los selectores (que entre otras cosas proporcionan una forma para escribir valores literales del tipo relevante), los operadores THE_, los operadores de asignación y comparación por igualdad, así como los operadores para prueba de tipo (vea el capítulo 19). *Nota:* Sin embargo, rechazamos las "funciones constructoras". Los constructores forman *variables;* puesto que la única variable que queremos en la base de datos es específicamente la varrel, el único "constructor" que necesitamos es un operador que crea una varrel (CREATE TABLE o CREATE VIEW, en términos de SQL). Por el contrario, los selectores seleccionan *valores.* Asimismo, es obvio que los constructores también regresan *apuntadores hacia* las variables construidas, mientras que los selectores regresan los valores seleccionados en sí.
- **Mensajes.** De nuevo, el concepto es esencial, aunque preferiríamos usar el término más convencional *invocación* (y, de nuevo, evitar la noción de que tales invocaciones tienen que estar dirigidas hacia algún "objeto de destino" y en su lugar, tratar a todos lo argumentos por igual).
- **Jerarquía de clases** (y nociones relacionadas, herencia, sustituibilidad, polimorfismo de in clusión, etcétera). Necesaria pero ortogonal (vemos el soporte a jerarquías de clases —si es proporcionado— sólo como parte del soporte para las clases en sí).
- **Clase contra ejemplar contra colección.** Las distinciones son esenciales, por supuesto, pero ortogonales (los *conceptos* son distintos y en realidad es todo lo que hay que decir).
- **Vínculos.** Ya hemos argumentado en el capítulo 13 (sección 13.6) que no es una buena idea tratar a los "vínculos" como una construcción formalmente distinta; en especial si única mente son los vínculos binarios los que reciben tal tratamiento especial. Tampoco pensamos que sea buena idea tratar a las restricciones de integridad referencial asociadas en una forma que esté divorciada del tratamiento (en caso de haberlo) de las restricciones de inte gridad en general (vea más adelante).
- **Lenguaje integrado de programación de bases de datos.** Es bueno tenerlo, pero ortogo nal. Sin embargo, los lenguajes soportados realmente en los sistemas de objetos actuales son, por lo general, *de procedimientos* (3GL) y por lo tanto —argumentaríamos— es horrible tenerlos (de hecho, un gran paso hacia atrás).

Y ésta es una lista de las características que el "modelo de objetos" *no* soporta, por lo general, o no lo soporta bien:

- **Consultas *ad hoc.*** En general, los primeros sistemas de objetos no soportaban en absoluto las consultas *ad hoc.* Los sistemas más recientes sí lo hacen pero rompen generalmente la encapsulación o imponen límites a las consultas que pueden hacerse (lo que significa, en este último caso, que las consultas a fin de cuentas en realidad no son *ad hoc).*
- **Vistas.** Por lo general no son soportadas (básicamente por la misma razón por la que en ge neral no son soportadas las consultas *ad hoc). Nota:* Algunos sistemas de objetos soportan las variables de ejemplar "derivadas" o "virtuales" (necesariamente públicas); por ejemplo,

la variable de ejemplar EDAD podría ser derivada restando el valor de la variable de ejemplar FECHA_NACIMIENTO a la fecha actual. Sin embargo, dicha posibilidad queda corta con respecto al mecanismo de vistas completo, y en cualquier caso, ya hemos rechazado la noción de variables de ejemplar públicas.

- **Restricciones de integridad declarativas.** Por lo general no son soportadas (básicamente por las mismas razones por las que no son soportadas las consultas *ad hoc* ni las vistas). En realidad, generalmente no son soportadas aun en sistemas que sí soportan las consultas *ad hoc*.
- **Claves externas.** El "modelo de objetos" tiene varios mecanismos diferentes para el manejo de la integridad referencial y ninguno de ellos es muy similar al mecanismo de clave ex terna más uniforme del modelo relacional. Cuestiones como ON DELETE RESTRICT y ON DELETE CASCADE se dejan por lo general para el código procedural (probablemente métodos, posiblemente código de aplicación).
- **Cierre.** ¿Cuál es el equivalente de objetos para la propiedad de cierre relacional?
- **Catálogo.** ¿Dónde está el catálogo en un sistema de objetos? ¿A qué se parece? ¿Hay algún estándar? *Nota:* Estas cuestiones son por supuesto retóricas. Lo que sucede realmente es que el catálogo tiene que ser construido por el personal profesional cuyo trabajo es adaptar el DBMS de objetos para la aplicación en la que haya sido instalado, como dijimos al final de la sección 24.5. (Ese catálogo será entonces específico de la aplicación, al igual que el DBMS adaptado en conjunto.)

Para resumir, las buenas características (esenciales y fundamentales) del "modelo de objetos" —es decir, aquellas que realmente nos gustaría soportar— son las que muestra la siguiente tabla:

Característica	*Término preferido*	*Comentarios*
Clase de objetos	Tipo	Escalar y no escalar; posiblemente definida por el usuario
Objeto inmutable	Valor	Escalar y no escalar
Objeto mutable	Variable	Escalar y no escalar
Método	Operador	Incluyendo selectores, operadores THE_, ":=", "=" y operadores para prueba de tipo
Mensaje	Invocación a operador	Sin operando de "destino"

De manera más concisa, podríamos decir que la única buena idea de los sistemas de objetos en general, es el **soporte apropiado a los tipos de datos** (todo lo demás —incluyendo, en particular, la noción de *operadores definidos por el usuario*— se desprende de esa idea básica).* ¡Pero esa idea es difícilmente nueva!

*Algunas personas podrían decir que la herencia de tipos también es una buena idea. Estamos de acuerdo, pero mantenemos nuestra posición de que el soporte de la herencia es ortogonal al soporte para los objetos en sí.

EJERCICIOS

24.1 Defina los siguientes términos:

clase	método
encapsulation	objeto
función constructora	objeto de definición de clase
ID de objeto	variable de ejemplar privada
jerarquía de clases	variable de ejemplar protegida
jerarquía de contención	variable de ejemplar pública
mensaje	variables inversas

24.2 ¿Cuáles son las ventajas de los OIDs? ¿Cuáles las desventajas? ¿Cómo podrían ser implementados los OIDs?

24.3 En la sección 24.2 dimos dos formulaciones SQL de la consulta "obtener todos los rectángulos que traslapan al cuadrado unitario". Demuestre que esas dos formulaciones son equivalentes.

24.4 Investigue cualquier sistema de objetos que tenga disponible. ¿Qué lenguajes de programación soporta ese sistema? ¿Soporta un lenguaje de consulta? De ser así, ¿cuál es? En su opinión, ¿es más o menos poderoso que el SQL convencional? ¿A qué se parece el catálogo? ¿Cómo interroga el usua rio al catálogo? ¿Hay algún soporte para vistas? De ser así, ¿qué tan extenso es? (por ejemplo, ¿qué hay acerca de la actualización de vistas?) ¿Cómo se maneja la "información faltante"?

24.5 Diseñe una versión de objetos de la base de datos de proveedores y partes. *Nota:* Este diseño será usado como base para los ejercicios 24.6 a 24.8 a continuación.

24.6 Escriba un conjunto de declaraciones de definición de datos en OPAL para su versión de obje tos de proveedores y partes.

24.7 Delinee los detalles de los métodos de "población de la base de datos" necesarios para su ver sión de objetos de proveedores y partes.

24.8 Escriba código OPAL para las siguientes consultas contra su versión de objetos de proveedores y partes:

a. Obtener todos los proveedores de Londres.

b. Obtener todas las partes rojas.

24.9 Considere nuevamente la base de datos de educación. Muestre lo que está involucrado en:

a. Eliminar una inscripción.

b. Eliminar un empleado.

c. Eliminar un curso.

d. Deshacerse de la clase inscripciones.

e. Deshacerse de la clase empleados.

Puede dar por hecho que se aplica el proceso de recolección de basura estilo OPAL. Indique cualquier suposición que haga con respecto a asuntos tales como la eliminación en cascada, etcétera.

24.10 Suponga que una versión de objetos de la base de datos de proveedores, partes y proyectos va a ser representada por medio de una sola jerarquía de contención. ¿Cuántas jerarquías de éstas hay? ¿Cuál es la mejor?

24.11 Considere una variación de la base de datos de proveedores, partes y proyectos en la cual, en lugar de registrar que determinados proveedores proporcionan determinadas partes a determinados proyectos, deseamos registrar solamente que (a) determinados proveedores proporcionan determina das partes, (b) determinadas partes son proporcionadas a determinados proyectos y (c) determinados

proyectos son provistos por determinados proveedores. ¿Cuántos diseños de objetos posibles hay ahora (con o sin jerarquías de contención)?

24.12 Considere los factores de rendimiento que explicamos brevemente en la sección 24.5. ¿Son al gunos de ellos específicos de los objetos? Justifique su respuesta.

24.13 Por lo general, los sistemas de objetos soportan las restricciones de integridad en forma *proce dural* por medio de métodos; la excepción principal es que las restricciones *referenciales* son típicamente soportadas (al menos en parte) en forma *declarativa.* ¿Cuáles son las ventajas del soporte procedural? ¿Por qué piensa usted que las restricciones referenciales son manejadas en forma diferente?

24.14 Explique el concepto de *variables inversas.*

REFERENCIAS Y BIBLIOGRAFÍA

Las referencias [24.5], [24.7], [24.11], [24.15], [24.26], [24.31], [24.38], [24.41] y [24.50] son libros de texto sobre temas de objetos y cuestiones relacionadas. Las referencias [24.35], [24.36] y [24.52] son conjuntos de artículos de investigación. Las referencias [24.27], [24.28], [24.44] y [24.47] son tutoriales. Las referencias [24.4], [24.9], [24.23] y [24.37] describen sistemas específicos.

24.1 Malcolm Atkinson *et al.:* "The Object-Oriented Database System Manifesto", Proc. 1 st Int. Conf. on Deductive and Object-Oriented Databases, Kyoto, Japón (1989). Nueva York, N.Y.: Elsevier Science (1990).

Es uno de los primeros intentos para generar un consenso sobre lo que debe incluir "el modelo de objetos". Propone lo siguiente como características obligatorias; es decir, características que (de acuerdo con la opinión de los autores) deben ser soportadas para que el DBMS en cuestión se merezca el nombre de "orientado a objetos":

1. Colecciones
2. IDs de objetos
3. Encapsulación
4. Tipos o clases
5. Herencia
6. Enlace tardío
7. Compleción computacional
8. Tipos definidos por el usuario
9. Persistencia
10. Bases de datos grandes
11. Concurrencia
12. Recuperación
13. Consultas *ad hoc*

También explica determinadas características *opcionales,* que incluyen la herencia múltiple y la verificación de tipos en tiempo de compilación; determinadas características *abiertas,* que incluyen el "paradigma de programación"; y determinadas características sobre las cuales los autores no pudieron llegar a un consenso, que incluyen —de manera un poco sorpresiva, considerando su importancia— las vistas y las restricciones de integridad.

Nota: Las referencias [3.3] y [25.34] hacen comentarios sobre este artículo. Con relación a los comentarios que están en la referencia [3.3], dicho sea de paso, debemos decir que están basados en la premisa de que el objetivo del artículo es definir las características de un DBMS de propósito general bueno y genuino. No negamos que las características listadas anteriormente podrían ser útiles para un DBMS altamente especializado que esté atado a alguna aplicación específica, como CAD/CAM, donde no hay necesidad de (digamos) el soporte para restricciones de integridad; pero entonces nos preguntaríamos si tal sistema es en realidad un DBMS, tal como entendemos este término normalmente.

24.2 Malcolm P. Atkinson y O. Peter Buneman: "Types and Persistence in Database Programming Languages", *ACM Comp. Surv. 19,* No. 2 (junio, 1987).

Es uno de los primeros artículos, si no es que *el* primero, en establecer la posición de que la persistencia debe ser ortogonal al tipo. Este artículo es un buen punto inicial para las lecturas en el área de los lenguajes de programación de bases de datos en general (donde los "lenguajes de programación de bases de datos" son percibidos por muchos como el requisito de los sistemas de objetos; vea por ejemplo las referencias [24.11] y [24.12]).

24.3 Francois Bancilhon: "A Logic-Programming/Object-Oriented Cocktail", *ACM SIGMOD Record 15,* No. 3 (septiembre, 1986).

Para citar la introducción: "El enfoque orientado a objetos... parece estar particularmente bien adecuado para [manejar] nuevos tipos de aplicaciones tales como CAD, [ingeniería de] software e [inteligencia artificial]. Sin embargo, la extensión natural a la tecnología de base de datos relacional es... el paradigma de la programación lógica, [no] el orientado a objetos. [Este artículo trata el asunto de] si los dos paradigmas son compatibles". Y concluye, cautamente, que sí lo son. *Nota:* La referencia [24.49] proporciona un punto de vista opuesto.

24.4 J. Banerjee *et al:* "Data Model Issues for Object-Oriented Applications", *ACM TOOIS (Trans actions on Office Information Systems) 5,* No. 1 (marzo, 1987). Vuelto a publicar en Michael Stonebraker (ed.), *Readings in Database Systems* (2a. edición). San Mateo, Calif.: Morgan Kaufmann (1994). También vuelto a publicar en la referencia [24.52].

24.5 Douglas K. Barry: *The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases.* Nueva York, N.Y.: John Wiley and Sons (1996).

La tesis principal de este libro es que necesitamos un sistema de objetos, no uno relacional, si es que tenemos que manejar "datos complejos". Los datos complejos están caracterizados por (a) ser ubicuos, (b) carecer frecuentemente de una identificación única, (c) involucrar numerosos vínculos muchos a muchos y (d) requerir frecuentemente el uso de códigos de tipo "en el esquema relacional" (debido a la falta de un soporte directo para subtipos y supertipos en los productos SQL actuales). *Nota:* El autor es director ejecutivo del ODMG (Grupo de Administración de Datos de Objetos) [24.12],

24.6 David Beech: "A Foundation for Evolution from Relational to Object Databases", en J. W. Schmidt, S. Ceri y M. Missikoff (eds.), *Extending Database Technology.* Nueva York, N.Y.: Springer Verlag(1988).

Éste es uno de varios artículos que exponen la posibilidad de ampliar SQL para que se convierta en algún tipo de "SQL de objetos" u "OSQL" (sin embargo, le prevenimos que esos "SQL de objetos" a menudo no se parecen mucho al SQL convencional). La referencia [24.39] da más detalles sobre las propuestas de este artículo en particular.

24.7 Elisa Bertino y Lorenzo Martino: *Object-Oriented Database Systems: Concepts and Architec tures.* Reading, Mass.: Addison-Wesley (1993).

24.8 Anders Bjornerstedt y Christer Hultén: "Version Control in an Object-Oriented Architecture", en la referencia [24.36].

Muchas aplicaciones necesitan el concepto de **versiones** distintas de un objeto dado; ejemplos de tales aplicaciones incluyen el desarrollo de software, el diseño de hardware, la creación de documentos, etcétera. Y algunos sistemas de objetos soportan este concepto en forma directa (aunque en realidad es ortogonal a la pregunta de si estamos manejando un sistema de objetos o de algún otro tipo). Dicho soporte incluye, por lo general:

- La capacidad para crear una nueva versión de un objeto dado, **sacando** típicamente una copia del objeto y moviéndolo desde la base de datos hacia la estación de trabajo privada del usuario, donde puede ser conservado y modificarse a lo largo de un periodo posiblemente amplio (por ejemplo, horas o días).
- La capacidad para establecer una versión de objeto dada como la versión de base de datos ac tual, **registrándola** y moviéndola típicamente desde la estación de trabajo del usuario de re greso hacia la base de datos (la cual puede, a su vez, requerir algún tipo de mecanismo para **combinar** versiones distintas).

- La capacidad para **eliminar (y** tal vez **archivar)** versiones obsoletas.
- La capacidad para interrogar el **historial de versiones** de un objeto dado.

Observe que —como lo sugiere la figura 24.7— los historiales de versiones no son necesariamente lineales (la versión V.2 en la figura conduce a dos versiones distintas V.3a y V.3b, que son subsecuentemente mezcladas en una sola versión V.4).

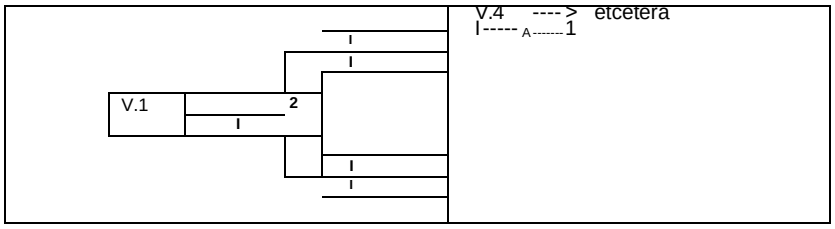


Figura 24.7 Historial típico de versiones.

Después, debido a que los objetos están típicamente interrelacionados en diversas formas, el concepto de versiones conduce al concepto de *configuraciones.* Una **configuración** es una colección de versiones de objetos interrelacionados. El soporte para configuraciones incluye, por lo general:

- La capacidad para **copiar** una versión de objeto desde una configuración hacia otra (por ejemplo, desde una configuración "vieja" hacia una "nueva").
- La habilidad para **mover** una versión de objeto desde una configuración hacia otra (es decir, añadirla a la configuración "nueva" y eliminarla de la "vieja").

De manera interna, dichas operaciones involucran básicamente mucho manejo de apuntadores, pero hay implicaciones mayores para la sintaxis del lenguaje y la semántica, en general, y para las consultas *adhoc* en particular. La mayoría de estas implicaciones están fuera del alcance de este libro, aunque es relevante el material del capítulo 22.

24.9 Paul Butterworth, Allen Otis y Jacob Stein: "The GemStone Object Database Management System", *CACM 34,* No. 10 (octubre, 1991).

24.10 Michael J. Carey, David J. DeWitt y Jeffrey F. Naughton: "The OO7 Object-Oriented Database Benchmark", Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data, Washington, DC (mayo, 1993).

24.11 R. G. G. Cattell: *Object Data Management* (edición revisada). Reading, Mass.: Addison-Wesley(1994).

Es el primer tutorial (con el tamaño de un libro) sobre la aplicación de la tecnología de objetos específicamente a las bases de datos. El siguiente extracto editado sugiere que el campo está todavía muy lejos de cualquier tipo de consenso: "es posible que los lenguajes de programación necesiten una nueva sintaxis... la estafa, la replicación y otros métodos nuevos de acceso también requieren mayor estudio... se requieren nuevas herramientas de desarrollo para aplicaciones y para usuarios finales... es necesario desarrollar características más poderosas para los lenguajes de consulta... se necesitan nuevas investigaciones sobre el control de la concurrencia... las marcas de tiempo y la semántica de la concurrencia basada en objetos necesitan más exploración... se necesitan modelos de rendimiento... es necesario integrar los nuevos trabajos sobre administración del conocimiento con las posibilidades de administración de datos y de objetos... esto [conducirá] a un problema de optimización complejo [y] pocos investigadores tienen la experiencia [necesaria]... las bases de datos federadas [de objetos] requieren mayor estudio".

24.12 R. G. G. Cattell y Douglas K. Barry (eds.): *The Object Database Standard: ODMG 2.0.* San Francisco, Calif.: Morgan Kaufmann (1997).

Usamos generalmente el término *ODMG* para referirnos a las propuestas del ODMG, un consorcio de representantes de "compañías [que cubren] casi toda la industria de los DBMSs de objetos". Estas propuestas consisten en un *Modelo de Objetos,* un *Lenguaje de Definición de Objetos,* un *Formato de Intercambio de Objetos,* un *Lenguaje de Consulta de Objetos* y *enlaces* entre estas propiedades y C++, Smalltalk y Java. (No hay componente de "lenguaje de manipulación de objetos"; en su lugar, las posibilidades de manipulación de objetos son proporcionadas por el lenguaje al que esté vinculado el ODMG.)

Puede encontrar un análisis detallado y crítico del modelo de objetos ODMG en la referencia [3.3]. Vea también la referencia [24.34].

24.13 R. G. G. Cattell y J. Skeen: "Object Operations Benchmark", *ACM TODS 17,* No. 1 (marzo, 1992).

24.14 George Copeland y David Maier: "Making Smalltalk a Database System", Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (junio, 1984). Vuelto a publicar en Michael Stonebraker (ed.), *Readings in Database Systems* (2a. edición). San Mateo, Calif.: Morgan Kaufmann (1994).

Describe algunas de las mejoras y cambios que se hicieron a Smalltalk [24.26] para crear GemStone y OPAL.

24.15 Brad J. Cox: *Object Oriented Programming: An Evolutionary Approach.* Reading, Mass.: Addison-Wesley (1986).

Es un texto tutorial sobre ideas de objetos dentro del mundo de la programación, con énfasis en el uso de técnicas de objetos para la ingeniería de software.

24.16 O. J. Dahl, B. Myhrhaug y K. Nygaard: *The SIMULA 67 Common Base Language,* Pub. S-22, Norwegian Computing Center, Oslo, Noruega (1970).

SIMULA 67 fue un lenguaje diseñado expresamente para la escritura de aplicaciones de simulación. Los lenguajes de programación de objetos crecieron a partir de dichos lenguajes; en realidad, SIMULA 67 fue verdaderamente el primer lenguaje de objetos.

24.17 C. J. Date: "An Optimization Problem", en C. J. Date y Hugh Darwen, *Relational Database Writings 1989-1991.* Reading, Mass.: Addison-Wesley (1992).

24.18 C. J. Date: "Why the 'Object Model' Is Not a Data Model", en C. J. Date, Hugh Darwen y David McGoveran, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).

24.19 C. J. Date: "Object Identifiers *vs.* Relational Keys," en C. J. Date, Hugh Darwen, and David McGoveran, *Relational Database Writings ¡994-1997.* Reading, Mass.: Addison-Wesley (1998).

24.20 C. J. Date: "Encapsulation Is a Red Herring", *DBP&D 12,* No. 9 (septiembre, 1998).

En el cuerpo de este capítulo dijimos que la encapsulacion implica la independencia de datos. Sin embargo, también dijimos que preferiríamos no usar el término "encapsulacion" (sino el término *escalar).* Parte del asunto es que los "objetos encapsulados" no pueden proporcionar mayor independencia de datos de lo que pueden las relaciones *no encapsuladas* (al menos en principio). Por ejemplo, no hay una razón absoluta por la cual una relación base que representa puntos, con atributos de coordenadas cartesianas X y Y, no deba ser guardada, en su lugar, en términos de coordenadas polares R y 0.

24.21 C. J. Date: "Persistence *Not* Orthogonal to Type", en el sitio web *DBP&D www.dbpd.com* (oc tubre, 1998).

24.22 C. J. Date: "Decent Exposure", en el sitio web *DBP&D www.dbpd.com* (noviembre, 1998).

24.23 O. Deux *et al:* "The O2 System", *CACM 34,* No. 10 (octubre, 1991).

24.24 Fabrizio Ferrandina, Thorsten Meyer, Roberto Zicari, Guy Ferran y Joelle Madec: "Schema and Database Evolution in the O2 Object Database System", Proc. 21st Int. Conf. on Very Large Data Bases, Zurich, Suiza (septiembre, 1995).

Vea el comentario a la referencia [24.43].

24.25 Jürgen Frohn, Georg Lausen y Heinz Uphoff: "Access to Objects by Path Expressions and Rules", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

24.26 Adele Goldberg y David Robson: *Smalltalk-80: The Language and Its Implementation.* Read ing, Mass.: Addison-Wesley (1983).

El relato definitivo de los esfuerzos pioneros en el centro de investigación Xerox de Palo Alto para diseñar y construir el sistema Smalltalk-80. La primera parte del libro (que consta de cuatro partes) es una descripción detallada del lenguaje de programación Smalltalk-80, sobre el que está basado el lenguaje OPAL de GemStone.

24.27 Nathan Goodman: "Object-Oriented Database Systems", *InfoDB 4,* No. 3 (otoño, 1989).

Las primeras ediciones de este libro incluían la siguiente cita de este artículo: "En esta etapa es vano comparar los [enfoques] relacional y de objetos. Tenemos que comparar entre iguales: manzanas con manzanas, sueños con sueños, teorías con teorías y productos maduros con productos maduros... [El] enfoque relacional ha estado en existencia durante algún tiempo: se apoya en una base teórica muy sólida y es la base de un gran número de productos maduros. El enfoque de objetos, por el contrario, es nuevo (al menos en la arena de base de datos); no posee un fundamento teórico que pueda ser comparado significativamente con el modelo relacional y los pocos productos que existen difícilmente pueden ser descritos como maduros. Por lo tanto, todavía falta mucho por hacer antes de que podamos comenzar a considerar seriamente el asunto de si la tecnología de objetos llegará a representar una alternativa viable ante el enfoque relacional."

Aunque algunos de los comentarios anteriores todavía son aplicables, el tema se ha cristalizado en cierta forma desde que se publicaron esas ediciones anteriores. En muchas formas, ahora es posible ver las relaciones contra los objetos como algo parecido a la comparación de manzanas con naranjas, como veremos en el capítulo 25.

24.28 Nathan Goodman: "The Object Database Debate", "The Object Data Model" y "The Object Data Model in Action", *InfoDB 5,* No. 4 (invierno, 1990-91); *6,* No. 1 (primavera-verano, 1991); y *6,* No. 2 (otoño, 1991).

24.29 Nathan Goodman: "An Object-Oriented DBMS War Story: Developing a Genome Mapping Database in C++", en la referencia [24.35].

Este artículo soporta varias de las críticas que hacemos en el cuerpo del presente capítulo. Para citar el resumen: "En contra de la sabiduría convencional, nuestra experiencia sugiere que es un error hacer demasiado inteligente a la base de datos implementando programas complejos como métodos dentro de los objetos de la base de datos. Nuestra experiencia también indica que C++ es un lenguaje pobre para la implementación de bases de datos; tiene problemas relacionados con la mecánica de definición de atributos, la mecánica de referencia sistemática a objetos, la falta de un recolector de basura y otras trampas sutiles en el modelo de herencia. También encontramos que los DBMSs actuales basados en C++ carecen de funciones de bases de datos importantes, y para compensar esto nos vimos obligados a proporcionar nuestra propia implementación simple de las funciones DBMS estándar: registro de transacciones para recuperación hacia atrás y hacia delante, un monitor de transacciones de procesos múltiples, un lenguaje de consulta y un procesador de consultas, así como algunas estructuras de almacenamiento. De hecho, usamos el DBMS basado en C++

como un manejador de almacenamiento orientado a objetos y construimos sobre éste un sistema especializado de administración de datos para la elaboración del mapa del genoma en gran escala."

24.30 H. V. Jagadish y Xiaolei Qian: "Integrity Maintenance in an Object-Oriented Database", Proc. 18th Int. Conf. on Very Large Data Bases, Vancouver, Canadá (agosto, 1992).

Propone un mecanismo de integridad declarativo para los sistemas de objetos y muestra la manera en que un compilador de restricciones de integridad puede incorporar el código necesario para verificar la integridad en los métodos para las clases de objetos apropiadas.

24.31 David Jordan: C++ *Object Databases: Programming with the ODMG Standard.* Reading, Mass.: Addison-Wesley (1997).

24.32 Michael Kifer, Won Kim y Yehoshua Sagiv: "Querying Object-Oriented Databases", Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, Calif, (junio, 1992).

Propone otro "SQL de objetos" llamado XSQL.

24.33 Won Kim: "Object-Oriented Database Systems: Promises, Reality, and the Future", Proc. 19th Int. Conf. on Very Large Data Bases, Dublin, Irlanda (agosto, 1993).

24.34 Won Kim: "Observations on the ODMG-93 Proposal for an Object-Oriented Database Lan guage", *ACM SIGMOD Record 23,* No. 1 (marzo, 1994).

24.35 Won Kim (ed.): *Modern Database Systems: The Object Model, Interoperability, and Beyond.* Nueva York, N.Y.: ACM Press/Reading, Mass.: Addison-Wesley (1995).

24.36 Won Kim y Frederick H. Lochovsky: *Object-Oriented Concepts, Databases, and Applications.* Reading, Mass.: ACM Press/Addison-Wesley (1989).

24.37 Charles Lamb, Gordon Landis, Jack Orenstein y Dan Weinreb: "The ObjectStore Database Sys tem", *CACM34,* No. 10 (octubre, 1991).

24.38 Mary E. S. Loomis: *Object Databases: The Essentials.* Reading, Mass.: Addison-Wesley (1995).

24.39 Peter Lyngbaek *et al:* "OSQL: A Language for Object Databases", Technical Report HPL-DTD-91-4, Hewlett-Packard Company (enero 15, 1991).

Vea el comentario a la referencia [24.6].

24.40 Bertrand Meyer: "The Future of Object Technology", *IEEE Computer 31,* No. 1 (enero, 1998).

Para citar: "El futuro de las bases de datos [de objetos] es un tema interesante para la especulación... Los fabricantes de bases de datos relaciónales se las han arreglado desde 1986 para sofocar el crecimiento de las bases de datos [de objetos] mediante anuncios prioritarios... Diez años después, los expertos [en bases de datos de objetos] todavía le dirán que las ofertas de los principales fabricantes relaciónales... aún están muy lejos de lo que deberían ser... El mercado para las bases de datos [de objetos] continuará creciendo, pero se mantendrá como una fracción del mercado de bases de datos tradicional".

24.41 Kamran Parsaye, Mark Chignell, Setrag Koshafian y Harry Wong: *Intelligent Databases.* Nueva York, N.Y.: John Wiley & Sons (1989).

24.42 Alexandra Poulovassilis y Carol Small: "Investigation of Algebraic Query Optimisation for Database Programming Languages", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

24.43 John F. Roddick: "Schema Evolution in Database Systems—An Annotated Bibliography", *ACM SIGMOD Record 21,* No. 4 (diciembre, 1992).

Los productos de bases de datos tradicionales sólo soportan, por lo general, cambios simples a un esquema existente (por ejemplo, la incorporación de un nuevo atributo a una varrel base

existente). Sin embargo, algunas aplicaciones requieren de un soporte más sofisticado para cambios de esquema, y algunos prototipos de objetos han investigado este problema a profundidad. Observe que el problema es de hecho más complejo en un ambiente de objetos, debido a que el propio esquema lo es.

La siguiente taxonomía de posibles cambios de esquema está basada en una dada en un artículo sobre el prototipo de objetos ORION [24.4]. Hacemos notar que varias de ellas (¿cuáles?) parecen revelar alguna confusión sobre la distinción entre el modelo contra la implementación.

- Cambios a una clase de objetos
  1. Cambios a una variable de ejemplar
     - Agregar variable de ejemplar.
     - Eliminar variable de ejemplar.
     - Renombrar variable de ejemplar.
     - Cambiar el valor predeterminado de la variable de ejemplar.
     - Cambiar el tipo de datos de la variable de ejemplar.
     - Cambiar el origen de la herencia de la variable de ejemplar.
  2. Cambios a un método
     - Agregar método.
     - Eliminar método.
     - Renombrar método.
     - Cambiar el código interno del método.
     - Cambiar el origen de la herencia del método.
- Cambios a la jerarquía de clases (suponiendo herencia múltiple)
  - Agregar la clase A a la lista de superclases de la clase B.
  - Eliminar la clase A de la lista de superclases de la clase B.
- Cambios al esquema general
  - Agregar clase (en cualquier lugar).
  - Eliminar clase (en cualquier lugar).
  - Renombrar clase.
  - Dividir clase.
  - Fundir clases.

Sin embargo, no queda claro cuánta transparencia puede lograrse con respecto a los cambios anteriores, en especial debido a que las vistas generalmente no están soportadas. De hecho, la posibilidad de una "evolución del esquema" implica un problema importante para los sistemas de objetos, debido a su naturaleza 3GL. Como lo dice la referencia [25.34]: "si cambia la cantidad de índices o los datos son reorganizados para que estén agrupados de forma diferente, no hay forma de que [los métodos] aprovechen automáticamente dichos cambios".

Además, la evolución del esquema es también algo más que un *requerimiento* en los sistemas de objetos, debido a que muchas de las decisiones que señan decisiones del DBA —¡o incluso del DBMS!— en un sistema relacional, llegan a ser decisiones del programador de aplicaciones en un sistema de objetos (vea la referencia [24.44]). En particular, los ajustes al rendimiento pueden conducir a un rediseño del esquema (de nuevo, vea la referencia [24.44]).

24.44 C. M. Saracco: "Writing an Object DBMS Application" (en dos partes), *InfoDB* 7, No. 4 (in vierno, 1993-94) *eInfoDB8,* No. 1 (primavera, 1994).

Da algunos ejemplos de codificación simples pero muy completos (e informativos).

24.45 Gail M. Shaw y Stanley B. Zdonik: "A Query Algebra for Object-Oriented Databases", Proc. 6th IEEE Int. Conf. on Data Engineering (febrero, 1990). También: Technical Report TR CS-89-19, Dept. of Computer Science, Brown University, Providence, R.I. (marzo, 1989).

Este artículo sirve para soportar la aseveración del escritor en el sentido de que cualquier "álgebra de objetos" será inherentemente compleja (debido a que los objetos son complejos). En particular, la *igualdad* de objetos jerárquicos anidados arbitrariamente requiere de un tratamiento muy cuidadoso. La idea básica que está tras la propuesta específica de este artículo es que cada operador del álgebra de consulta produce una *relación*, en la cual cada tupia contiene OIDs para determinados objetos de la base de datos. En el caso de una junta, por ejemplo, cada tupia contiene OIDs de objetos que coinciden entre sí bajo la condición de junta. Estas tupias no heredan ningún método de los objetos componentes.

24.46 David W. Shipman: "The Functional Data Model and the Data Language DAPLEX", *ACM TODS 6*, No. 1 (marzo, 1981). Vuelto a publicar en Michael Stonebraker (ed.), *Readings in Database Systems* (2a. edición). San Mateo, Calif.: Morgan Kaufmann (1994).

Ha habido varios intentos a través de los años para construir sistemas basados *en funciones* en lugar de relaciones, y DAPLEX es uno de los más conocidos. La razón por la que lo mencionamos aquí es que los enfoques funcionales comparten determinadas ideas con el enfoque de objetos, incluyendo en particular un estilo con algunas características de navegación (es decir, trazado de ruta) para el direccionamiento de objetos que están relacionados funcionalmente con otros objetos (que a su vez están relacionados funcionalmente con otros objetos, etcétera). Observe, sin embargo, que una función en este contexto no es típicamente una función matemática en absoluto; podría ser, por ejemplo, multivaluada. En realidad, tenemos que ejercer una violencia considerable sobre el concepto de función para hacerlo capaz de todas las cosas que se requieren en el contexto del "modelo de datos funcional".

24.47 Jacob Stein y David Maier: "Concepts in Object-Oriented Data Management", *DBP&D 1,* No. 4 (abril, 1988).

Es una buena guía sobre conceptos de objetos escrito por dos diseñadores de GemStone.

24.48 D. C. Tsichritzis y O. M. Nierstrasz: "Directions in OO Research," en la referencia [24.36].

Este artículo da más peso a la aseveración de quien escribe esto en el sentido de que la tecnología de bases de datos de objetos está muy lejos de un consenso: "hay desacuerdos sobre definiciones básicas; por ejemplo, ¿qué es un objeto?... No hay razón para preocuparse; las definiciones am biguas son inevitables y a veces bienvenidas durante un periodo dinámico de descubrimiento científico. Deberán y llegarán a ser más rigurosas durante un periodo de consolidación que ine vitablemente vendrá a continuación". ¡Pero los conceptos de objetos ya llevan más de treinta años!; en realidad, son anteriores al modelo relacional.

24.49 Jeffrey D. Ullman: "A Comparison between Deductive and Object-Oriented Database Sys tems", Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, Munich, Germany (di ciembre, 1991); C. Delobel, M. Kifer y Y. Masunaga (eds.), *Lecture Notes in Computer Science 566.* Nueva York, N.Y.: Springer Verlag (1992).

Aunque no estamos de acuerdo con este artículo en varios puntos detallados, sí estamos de acuerdo con su conclusión general, la cual es que los sistemas de bases de datos "deductivos" (es decir, basados en la lógica) prometen más, a largo plazo, que los sistemas de objetos. También, el artículo tiene un punto importante con respecto a la *optimization.* Supongamos que fuéramos a definir "una clase de objetos que se comporte como las relaciones binarias, y...un método *join* para esta clase, a fin de que podamos escribir una expresión como R JOIN S JOIN T. Parece que podríamos evaluar esta expresión como (R JOIN S) JOIN T o como R JOIN (S JOIN T). ¿Pero podemos? Nunca se estableció lo que significa el método *join.* Por ejemplo, ¿es asociativo?... La conclusión es que, si queremos programar en forma [de objetos] y además programar al nivel de relaciones o superior, necesitamos darle al sistema la información que está comprendida en las...leyes del álgebra relacional. Esa información no puede ser deducida por el sistema, sino que debemos integrársela. Así,...la única parte del lenguaje de consulta que será optimizable es el conjunto de métodos fijos para los cuales...ha sido proporcionada la semántica adecuada al sistema".

24.50 Gottfried Vossen: *Data Models, Database Languages, and Database Management Systems.* Reading, Mass.: Addison-Wesley (1991).

24.51 Carlo Zaniolo: "The Database Language GEM", Proc. 1983 ACM SIGMOD Int. Conf. on Ma nagement of Data, San José, Calif, (mayo, 1983). Vuelto a publicar en Michael Stonebraker (ed.), *Readings in Database Systems* (2a. edición). San Mateo, Calif.: Morgan Kaufmann (1994).

GEM significa "manipulador general de entidades ". Es en efecto una extensión de QUEL que soporta relaciones con atributos con valores de conjunto y relaciones con atributos alternos (es decir, los EMPs exentos pueden tener un atributo SALARIO, mientras que los no exentos pueden tener TARIFA_HORA y TIEMPO_EXTRA). También utiliza la idea de que los objetos contienen conceptualmente a otros objetos (en lugar de claves externas que hacen referencia a esos otros objetos), y extiende la notación familiar de calificación mediante puntos, a fin de proporcionar una forma simple para hacer referencia a los atributos de esos objetos contenidos; en realidad, recorre implícitamente determinadas rutas de junta predefinidas. Por ejemplo, podríamos usar el nombre calificado EMP.DEPTO.PRESUPUESTO para referirnos al presupuesto del departamento de algún empleado dado. Muchos sistemas han adoptado o adaptado esta idea.

24.52 Stanley B. Zdonik y David Maier (eds.): *Readings in Object-Oriented Database Systems.* San Francisco, Calif.: Morgan Kaufmann (1990).

RESPUESTAS A EJERCICIOS SELECCIONADOS

24.1 Aquí (sólo) comentamos el propio término *objeto.* Éstas son algunas "definiciones" encontradas en la literatura:

- "Los objetos son módulos de código reutilizables que guardan datos, información sobre víncu los entre los datos y las aplicaciones, así como procesos que controlan los datos y los vínculos" (tomado de un anuncio de un producto comercial).
- "Un objeto es un bloque de memoria privada con una interfaz pública" [24.47].
- "Un objeto es una máquina abstracta que define un protocolo mediante el cual los usuarios del objeto pueden interactuar " (introducción de la referencia [24.52]).
- "Un objeto es una estructura de software que contiene datos y programas" [24.27].
- "Todo es un objeto... Un objeto tiene una memoria privada y una interfaz pública [24.50].

Observe que *ninguna* de estas "definiciones" obtiene lo que podríamos considerar como el punto medular de la materia; es decir, que un objeto es esencialmente sólo un *valor* (cuando es inmutable) o una *variable* (cuando no lo es).

También vale la pena comentar la noción de que "todo es un objeto". Éstas son algunas construcciones de ejemplo que no son objetos en la mayoría de los sistemas de objetos: *variables de ejemplar, vínculos* (al menos en ODMG [24.12]), *métodos, OIDs, variables de programa.* Y en algunos sistemas (incluyendo nuevamente a ODMG) los *valores* tampoco son objetos.

24.2 Algunas de las ventajas de los OIDs son las siguientes:

- No son "inteligentes". Vea la referencia [13.9] para una explicación de por qué es necesaria esta situación.
- Nunca cambian mientras el objeto que identifican siga existiendo.
- No son compuestos. Vea las referencias [8.12], [13.10] y [18.8] para una explicación de por qué es necesaria esta situación.
- Todo en la base de datos está identificado de la misma forma uniforme (a diferencia de la situación en las bases de datos relaciónales).
- No hay necesidad de repetir claves de usuario al hacer referencia a objetos. Por lo tanto, no es necesaria ninguna regla ON UPDATE.

Algunas de las ¿fesventajas —que no impiden la necesidad de claves de usuario, que conducen a un estilo de programación de "caza de apuntadores" de bajo nivel y que se aplican solamente a objetos "base" (no derivados)— las tratamos brevemente en las secciones 24.2 a 24.4.

Las técnicas de implementación de OIDs posibles incluyen:

- Direcciones físicas de disco (son rápidas pero dan una independencia de datos pobre).
- Direcciones lógicas de disco (es decir, direcciones de página y desplazamiento; son bastante rápi das y dan una mejor independencia de datos).
- IDs artificiales (por ejemplo, marcas de tiempo, números de secuencia; necesitan transformación hacia las direcciones reales).

24.3 Vea la referencia [24.17].

24.5 No damos una respuesta detallada a este ejercicio, sino que proporcionamos algunos comentarios sobre la cuestión del diseño de bases de datos de objetos, en general. En ocasiones decimos que los sistemas de objetos facilitan el diseño de base de datos (así como su uso), debido a que proporcionan construcciones de modelado de alto nivel y soportan esas construcciones directamente en el sistema. (Por el contrario, los sistemas relaciónales involucran un nivel de indirección adicional; es decir, el proceso de correspondencia entre los objetos reales y las varrels, atributos, claves externas, etcétera). Y hay algún mérito en este comentario. Sin embargo, evade la gran pregunta: ¿en primer lugar, cómo se hace el diseño de una base de datos de objetos? El hecho es que "el modelo de objetos", como es entendido generalmente, involucra muchos más *grados de libertad*—en otras palabras, más alternativas— que como sucede en el modelo relacional; y en mi opinión, no creo que exista un buen lineamiento que ayude a seleccionar entre esas alternativas. Por ejemplo, ¿cómo expresamos la manera de representar, digamos, el conjunto de todos los empleados (como un arreglo, una lista, un conjunto, etcétera)? "Un modelo de datos poderoso necesita una metodología de diseño poderosa... y ésta es una *desventaja* del modelo de objetos" (si parafraseamos un poco la referencia [24.27], podríamos argumentar que el calificativo "poderoso" en realidad debería ser "complicado").

24.9 No damos una respuesta detallada a este ejercicio, pero hacemos un comentario con respecto a su dificultad. Primero, acordemos usar el término "eliminar" como una abreviatura que significa "hacer un candidato para la eliminación física" (es decir, borrar todas las referencias hacia el objeto en cuestión). Luego, para eliminar un objeto *X*, primero debemos encontrar todos los objetos *Y* que incluyen una referencia hacia *X;* para cada objeto *Y* de éstos, debemos entonces eliminar ese objeto y, o al menos borrar la referencia que está en ese objeto *Y* hacia el objeto *X* (asignando a esa referencia el valor especial *nil).* Y parte del problema es que no es posible decir —a partir únicamente de la definición de datos— exactamente qué objetos incluyen una referencia hacia *X* (y ni siquiera qué tantos de ellos hay). Considere por ejemplo a los empleados y la clase de objetos ESET. En principio, podría haber cualquier cantidad de ejemplares de ESET, y cualquier subconjunto de esos ejemplares ESET podría incluir una referencia hacia algún empleado específico.

24.10 Hay obviamente seis jerarquías posibles:

```
V contiene ( P contiene ( Y ) )
V contiene ( Y contiene ( P ) )
P contiene ( Y contiene ( V ) )
P contiene ( V contiene ( Y ) )
Y contiene ( V contiene ( P ) )
Y contiene ( P contiene ( V ) )
```

No es posible responder ¿"cuál es la mejor?" sin información adicional, pero casi seguramente *todas* ellas son malas.

24.11 Hay al menos doce diseños "obvios" de jerarquía de contención. Estos son cuatro de ellos:

```
V contiene ( P contiene ( Y ) )
V contiene ( Y contiene ( P ) )
V contiene ( primero P luego Y )
V contiene ( primero Y luego P )
```

Hay muchos otros diseños candidatos; por ejemplo, una clase "VP" que muestre directamente cuáles proveedores proporcionan cuáles partes, incluyendo también dos conjuntos de proyectos incrustados, uno para el proveedor y otro para la parte.

También hay un diseño muy simple que no involucra ninguna jerarquía (no trivial) y que consiste en una clase "VP", una clase "PY" y una clase "YV".

24.12 Los factores de rendimiento que tratamos fueron *agrupamiento, uso de memoria caché, estafa de apuntadores* y *ejecución de métodos en el servidor*. Todas estas técnicas son aplicables a cualquier sistema que proporciona un nivel suficiente de independencia de datos; por lo tanto, en realidad no son "específicas de los objetos". De hecho, la idea de usar la definición de base de datos lógica para decidir qué agrupamiento físico usar, como lo hacen algunos sistemas de objetos, podría verse como el socavamiento potencial de la independencia de datos. *Nota:* También debemos señalar que otro fac tor de rendimiento muy importante (la **optimization)** por lo general *no* se aplica en los sistemas de objetos.

24.13 En mi opinión, el soporte declarativo, en caso de ser posible, *siempre* es mejor que el soporte procedural (en todos los aspectos y no sólo por las restricciones de integridad). En pocas palabras, el soporte declarativo significa que el sistema hace el trabajo en lugar del usuario. Ésta es la razón por la cual los sistemas relaciónales soportan las consultas declarativas, las definiciones de vistas declara tivas, las restricciones de integridad declarativas, etcétera.

CAPITULO 25

Bases de datos de objetos/relaciónales

25.1 INTRODUCCIÓN

En los cinco años que han transcurrido desde que se publicó la edición anterior de este libro, varios fabricantes han lanzado productos DBMS "de objetos/relaciónales" (también conocidos, al menos en el mercado, como *servidores universales).* Ejemplos de ellos incluyen la versión Universal Database de DB2, Universal Data Option para Informix Dynamic Server y Oracle 8i Universal Server, Database Server o Enterprise Server (parece que se usan todos estos nombres). La idea general en cada caso, es que los productos deben soportar posibilidades de objetos y relaciónales; en otras palabras, los productos en cuestión son intentos de un *acercamiento* entre las dos tecnologías.

Ahora, en mi opinión, cualquiera de estos *acercamientos* deberá estar basado firmemente en el modelo relacional (que es a final de cuentas, la base de la tecnología moderna de bases de datos en general, como explicamos en la parte II de este libro). Por lo tanto, lo que queremos es que los sistemas relaciónales evolucionen* para que incorporen las características de los objetos, o al menos las buenas características (seguramente no queremos descartar completamente a los sistemas relaciónales ni queremos tratar con dos sistemas independientes, el relacional y el de objetos, en forma paralela). Y esta opinión es compartida por muchos otros escritores, incluyendo en particular a los autores del Manifiesto de los Sistemas de Bases de Datos de Tercera Generación [25.34], quienes establecen categóricamente que **los DBMSs de tercera generación deben incluir a los DBMSs de segunda generación** (donde los "DBMSs de primera generación" son los prerrelacionales, los "DBMSs de segunda generación" son los sistemas SQL y los "DBMSs de tercera generación" son lo que vengan después). Sin embargo, esta opinión *no* es compartida —aparentemente— por algunos de los productos de objetos ni por determinados escritores de objetos. Esta es una cita típica:

> La ciencia de la computación ha visto muchas generaciones de administración de datos, comenzando con los archivos indexados y posteriormente con DBMSs de red y jerárquicos... [y] más recientemente los DBMSs relaciónales... Ahora, estamos al borde de otra generación de sistemas de bases de datos... [que] proporcionan *administración de objetos,* [que soportan] tipos de datos mucho más complejos [25.4].

---

* Observe que estamos interesados definitivamente en la evolución y no en la revolución. Por el contrario, considere esta cita del libro de ODMG [24.13]: "[los DBMSs de objetos] representan *un desarrollo revolucionario antes que uno evolucionista"* (agregamos las cursivas). No creemos que el mercado en general esté listo para la revolución, ni pensamos que lo necesite o que debería estarlo; ésta es una razón por la cual *El Tercer Manifiesto* [3.3] es de manera específica muy evolucionista, no revolucionario, por naturaleza.

Aquí el escritor sugiere claramente que, así como los sistemas relaciónales desplazaron a los sistemas jerárquicos y de red antiguos, también los sistemas de objetos desplazarán a su vez a los sistemas relaciónales.

La razón por la que no estamos de acuerdo con esta posición es que el sistema *relacional es realmente diferente* [25.13]. Es diferente porque no es *ad hoc*. Los sistemas prerrelacionales antiguos eran *ad hoc*\ pudieron haber proporcionado soluciones para determinados problemas importantes en sus días, pero no se apoyaban en ninguna base teórica sólida. Por desgracia, los partidarios del sistema relacional —incluyéndome a mí— se vieron inicialmente perjudicados cuando argumentaron los méritos relativos de los sistemas relaciónales y prerrelacionales; por supuesto, tales argumentos fueron necesarios en ese momento, pero tuvieron el efecto imprevisto de reforzar la idea de que los DBMSs relaciónales y prerrelacionales eran esencialmente el mismo tipo de cosa. Y esta idea errónea soporta a su vez la posición que mencionamos anteriormente; es decir, que los objetos son para las relaciones lo que las relaciones fueron para las jerarquías y las redes.

Entonces, ¿qué hay acerca de los objetos? ¿Son *ad hoc!* La siguiente cita tomada del Manifiesto de los Sistemas de Bases de Datos Orientadas a Objetos [24.1] es reveladora en este aspecto: "Con respecto a la especificación del sistema estamos tomando un enfoque darwiniano. Esperamos que después del conjunto de prototipos experimentales que están siendo construidos, emergerá un modelo [de objetos] adecuado". En otras palabras, la sugerencia es aparentemente que ¡debemos escribir código y construir sistemas ***sin*** ningún modelo predefinido y ver qué sucede!

Por lo tanto, en lo que sigue a continuación tomamos como axioma —y a propósito, así lo hace la mayoría de los fabricantes de los principales DBMSs— que lo que queremos hacer es mejorar los sistemas relaciónales para incorporar las mejores características de la tecnología de objetos. Para repetir, *no* queremos descartar la tecnología relacional; sería muy lastimoso desandar el camino de más de treinta años de investigación y desarrollo relaciónales sólidos.

Ahora bien, en el capítulo 24 argumentábamos —vea también el comentario a la referencia [25.23]— que la orientación a objetos involucraba solamente una buena idea, específicamente el **soporte apropiado a los tipos de datos** (o dos buenas ideas, si es que quiere contar por separado la herencia de tipos). Por lo tanto, la pregunta que tenemos por delante se convierte en: ¿Cómo podemos incorporar al modelo relacional el soporte apropiado para los tipos de datos? Y la respuesta es, como probablemente ya se haya dado cuenta, que el soporte ya está ahí en la forma de *dominios* (a los cuales preferimos llamar *tipos).* En otras palabras, no necesitamos hacer **nada** al modelo relacional para lograr la funcionalidad de objetos en los sistemas relaciónales; es decir, nada además de implementarlo, completa y adecuadamente, y es ahí donde la mayoría de los sistemas actuales han fallado notablemente.*

Por lo tanto, creemos que un sistema relacional que soporte adecuadamente los dominios, sería capaz de manejar todos estos tipos de datos "problemáticos" que (según se dice a veces) pueden ser manejados por los sistemas de objetos pero no por los relaciónales. Estos datos son:

---

*En particular, esos sistemas han dado lugar a la idea falsa (demasiado común) de que los sistemas relaciónales sólo pueden soportar una cantidad limitada de tipos muy simples. Las siguientes citas son típicas: "los sistemas de bases de datos relaciónales soportan una colección pequeña y fija de tipos de datos" [25.25]; "un DBMS relacional sólo puede soportar... sus tipos integrados [básicamente sólo números, cadenas, fechas y horas]" [24.38]; "los *modelos de datos de objetos/relaciónales* extienden el modelo de datos relacional proporcionando un sistema de tipos más rico" [17.61]; etcétera.

datos de series de tiempo, datos biológicos, datos financieros, datos de diseño de ingeniería, datos de automatización de oficina, etcétera. En forma similar, también creemos que un sistema "de objetos/relacional" verdadero sería ni más ni menos un sistema *relacional* verdadero; es decir, un sistema que soporta el modelo relacional, con todo lo que implica dicho soporte. Por lo tanto, los fabricantes de DBMS deberían ser motivados para que hicieran lo que de hecho están tratando de hacer: precisamente extender sus sistemas para que incluyan el soporte apropiado de tipos o dominios. De hecho, podemos argumentar que la razón principal por la cual los sistemas de objetos han sido atractivos es precisamente debido a que los fabricantes de SQL no han soportado adecuadamente al modelo relacional. Pero este hecho no debe ser visto como un argumento para abandonar completamente los sistemas relaciónales (¡en absoluto!).

A manera de ejemplo, ahora nos encargaremos de algunas cosas que no terminamos en el capítulo 24 (sección 24.1) y mostraremos una buena solución *relacional* para el problema de los rectángulos. Esta solución implica primero la definición de un *tipo* rectángulo (digamos RECT):

```
TYPE RECT POSSREP ( X1 RATIONAL, Y1 RATIONAL,
                    X2 RATIONAL, Y2 RATIONAL ) ... ;
```

Damos por hecho que los rectángulos están representados *físicamente* por medio de una de esas estructuras de almacenamiento que están orientadas específicamente para soportar eficientemente los datos espaciales; quadtrees, árboles-R, etcétera [25.27].

También definimos un operador para probar si dos rectángulos dados se traslapan:

```
OPERATOR TRASLAPA  ( R1 RECT, R2 RECT )
   RETURNS ( BOOLEAN ) ;
   RETURN ( THE_X1(R1) < THE_X2(R2)  AND
            THE_Y1(R1) < THE_Y2(R2)  AND
            THE_X2(R1) >THE_X1(R2)  AND
            THE_Y2(R1) >THE_Y1(R2)   ) ;
END OPERATOR ;
```

Este operador implementa la forma "corta" *eficiente* de la prueba de traslape (consulte el capítulo 24 si necesita recordar a lo que se refiere esta forma corta) contra la estructura de almacenamiento *eficiente* (árbol-R o cualquier otra cosa).

Ahora el usuario puede crear una varrel base con un atributo de tipo RECT:

```
VAR RECTÁNGULO RELATION { R RECT, ... } KEY { R } ¡
```

Y la consulta "obtener todos lo rectángulos que traslapan al cuadrado unitario" se convierte ahora simplemente en:

```
RECTÁNGULO WHERE TRASLAPA ( R, RECT ( 0.0, 0.0, 1.0, 1.0 ) )
```

Esta solución resuelve *todas* las desventajas que tratamos en el capítulo 24 con relación a esta consulta. |

El plan del resto del capítulo es el siguiente. Las secciones 25.2 y 25.3 tratan *los dos grandes errores garrafales* (al menos uno que está siendo cometido aparentemente por casi todos los productos de objetos/relaciónales que hay en el mercado). Luego, la sección 25.4 considera ciertos aspectos de la implementación de un sistema de objetos/relacional. La sección 25.5 describe los beneficios de un sistema de objetos/relacional *genuino* (es decir, uno que no comete ninguno de estos dos errores garrafales). La sección 25.6 proporciona un resumen.

25.2 EL PRIMER GRAN ERROR GARRAFAL

Comenzamos con una cita de la referencia [3.3]:

> [Antes] que podamos considerar la cuestión de la integración de objetos y relaciones en cualquier nivel de detalle, hay un asunto preliminar crucial que debemos tratar y es el siguiente:
>
> **¿Qué concepto hay en el mundo relacional que sea la contraparte del concepto clase de *objetos* en el mundo de los objetos?**
>
> La razón por la cual esta pregunta es tan crucial es que, en realidad, la *clase de objetos* es el concepto más fundamental en todo el mundo de los objetos; todos los demás conceptos de objetos dependen de él en mayor o menor grado. Y hay dos ecuaciones que pueden ser, y han sido, propuestas como respuestas a esta pregunta:
>
> - dominio = clase de objetos,
> - varrel = clase de objetos.
>
> Ahora argumentaremos fuertemente que la primera de estas ecuaciones es correcta y la segunda está equivocada.

En realidad, la primera ecuación es *obviamente* correcta, debido a que las clases de objetos y los dominios son simplemente tipos. De hecho, si tomamos en cuenta que las varrels son *variables* y las clases son *tipos*, debería también ser inmediatamente obvio que la segunda ecuación es errónea (las variables y los tipos no son lo mismo); por esta razón, *El tercer manifiesto* [3.3] asevera categóricamente que **las varrels no son dominios.** Sin embargo, mucha gente y algunos productos se han apegado de hecho a la segunda ecuación; un error al que llamamos **el gran error garrafal** (o más precisamente, El *primer* gran error garrafal, ya que veremos otro dentro de un momento). Por lo tanto, es instructivo dar una mirada muy cuidadosa a la segunda ecuación y así lo hacemos. *Nota:* La mayor parte del resto de esta sección está tomada, más o menos al pie de la letra, de la referencia [3.3].

¿Por qué alguien podría cometer tal error garrafal? Bien, considere la siguiente definición de clase simple, expresada en un lenguaje de objetos hipotético que es similar, pero deliberadamente no idéntico, al de la sección 24.3:

```
                CREATE OBJECT CLASS EMP
EMP#          CHAR(5),
NOMEMP        CHAR(20)
SAL           NUMERIC,
PASATIEMPO    CHAR (20)
TRABAJA_PARA  CHAR(20)
```

(Aquí EMP#, NOMEMP, etcétera, son *variables de ejemplar públicas.)* Y ahora considere la siguiente definición SQL de "varrel base":

```
CREATE TABLE EMP
    ( EMP#            CHAR(5),
      NOMEMP          CHAR(20)
      SAL             NUMERIC,
      PASATIEMPO      CHAR(20)
      TRABAJA PARA    CHAR(20)
```

Estas dos definiciones en realidad se ven muy similares y la idea de igualarlas se ve muy tentadora. Y (como ya dijimos) determinados sistemas, incluyendo algunos productos comerciales, han hecho simplemente eso. Entonces, veámoslo más de cerca. Más precisamente, tomemos la instrucción CREATE TABLE que acabamos de mostrar y consideremos una serie de extensiones posibles que (dirían algunos) sirven para hacerla más "parecida a los objetos". *Nota:* La explicación que viene a continuación está basada en un producto comercial específico; en realidad, está basada en un ejemplo de la documentación propia de ese producto. Sin embargo, aquí no identificamos ese producto debido a que no pretendemos en este libro criticar o alabar productos específicos. En vez de ello, las críticas que haremos posteriormente en esta sección se aplican, haciendo los cambios necesarios, a *cualquier* sistema que se adhiera a la ecuación "varrel = clase".

La primera extensión es permitir *atributos compuestos* (es decir, con valores de tupia); esto es, permitimos que los valores de atributos sean tupias de alguna otra varrel (o posiblemente de la misma varrel). En el ejemplo, podemos reemplazar la instrucción CREATE TABLE original por la siguiente colección de declaraciones (consulte la figura 25.1):

```
CREATE TABLE EMP
    ( EMP#           CHAR(5),
      NOMEMP         CHAR(20),
      SAL            NUMERIC,
      PASATIEMPO     ACTIVIDAD,
      TRABAJA PARA   COMPAÑÍA )

                CREATE TABLE ACTIVIDAD
                     ( NOMBRE        CHAR(20),
                       EQUIPO       INTEGER ) ;
                CREATE TABLE COMPAÑÍA
                     ( NOMBRE        CHAR(20),
                       UBICACIÓN     CIUDADESTADO )
                CREATE TABLE CIUDADESTADO
                     ( CIUDAD       CHAR(20),
                       ESTADO       CHAR(2) ) ;
```

Explicación: El atributo PASATIEMPO en la varrel EMP está declarado para que sea de tipo ACTIVIDAD. ACTIVIDAD, a su vez, es una varrel con dos atributos, NOMBRE y EQUIPO, donde EQUIPO da la cantidad de jugadores en el equipo que corresponde a NOMBRE; por ejemplo, una "actividad" posible podría ser (Fútbol, 11). Entonces, cada valor de PASATIEMPO en

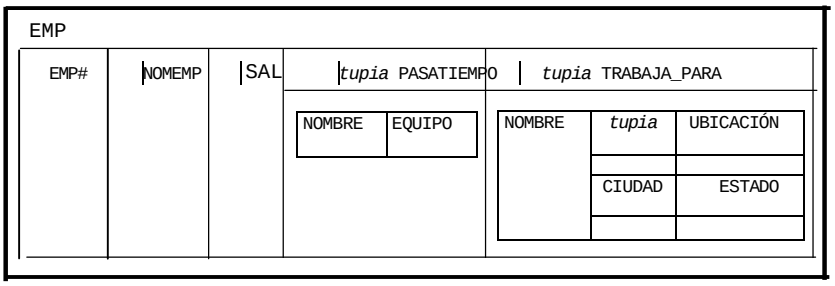


Figura 25.1 Atributos que contienen (apuntadores a) tupias; desaprobado.

realidad es un *par* de valores, un valor NOMBRE y un valor EQUIPO (más precisamente, es un par de valores que aparece actualmente como una tupia en la varrel ACTIVIDAD). *Nota:* Observe que ya hemos violado lo que manda el *Tercer Manifiesto* de que las varrels no son dominios; el "dominio" para el atributo PASATIEMPO está definido para que sea la *varrel* ACTIVIDAD. Para una mejor explicación de este punto, vea más adelante en esta sección.

En forma similar, el atributo TRABAJA_PARA en la varrel EMP está declarado para que sea de tipo COMPAÑÍA, y COMPAÑÍA también es una varrel de dos atributos, uno de ellos definido para que sea del tipo CIUD ADESTADO, el cual es otra varrel de dos atributos, y así sucesivamente. En otras palabras, las varrels ACTIVIDAD, COMPAÑÍA y CIUD ADESTADO son consideradas como *tipos* (o dominios) así como varrels. Por supuesto, lo mismo es cierto para la propia varrel EMP.

Por lo tanto, esta primera extensión es, por así decirlo, equivalente a permitir que los objetos contengan otros objetos y por lo tanto, a soportar el concepto *de jerarquía de contención* (vea el capítulo 24).

Además, insistimos en que hemos caracterizado esta primera extensión como "atributos que contienen tupias", debido a que ésta es la forma en que los partidarios de la ecuación "varrel = clase" la caracterizan (vea por ejemplo la referencia [24.33]). Sin embargo, sería más preciso caracterizarla como "atributos que contienen *apuntadores hacia* tupias"; un aspecto que examinaremos dentro de unos momentos. (Por lo tanto, en la figura 25.1 deberíamos reemplazar realmente cada una de las tres apariciones del término *tupia* por el término *apuntador hacia tupia.)*

Consideraciones similares a las de estos párrafos, se aplican también a la segunda extensión, que es permitir los *atributos con valor de relación;* es decir, permitir que los valores de atributos sean *conjuntos* de tupias de alguna otra varrel (o posiblemente de la misma varrel). Por ejemplo, suponga que los empleados pueden tener una cantidad cualquiera de pasatiempos en lugar de sólo uno (consulte la figura 25.2):

```
CREATE TABLE EMP
    ( EMP#            CHAR(5),
      NOMEMP          CHAR(20),
      SAL             NUMERIC,
      PASATIEMPOS     SET OF ( ACTIVIDAD ),
      TRABAJA PARA    COMPAÑÍA ) :
```

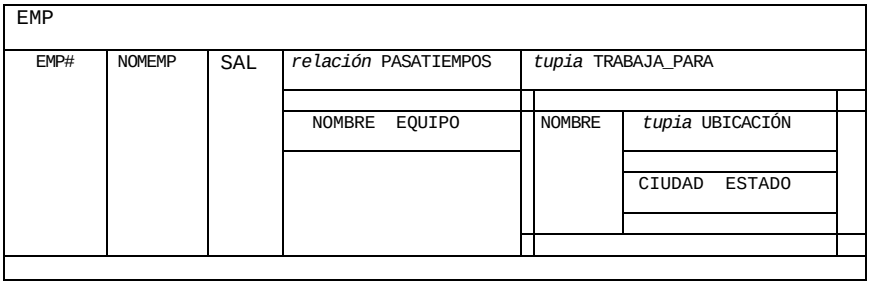


Figura 25.2 Atributos que contienen conjuntos de (apuntadores hacia) tupias; desaprobado.

Explicación: El valor PASATIEMPOS dentro de cualquier tupia dada de la varrel EMP, es ahora (conceptualmente) un conjunto de cero o más pares (NOMBRE, EQUIPO) —es decir, tupias— de la varrel ACTIVIDAD. Por lo tanto, esta segunda extensión es, por así decirlo, equivalente a permitir que los objetos contengan objetos de "colección": una versión más compleja de la jerarquía de contención. *Nota:* De paso, comentamos que en el producto particular sobre el cual está basado nuestro ejemplo, esos objetos de colección pueden ser *secuencias* o *bolsas*, así como conjuntos en sí.

La tercera extensión es permitir que las varrels tengan *métodos* asociados. Por ejemplo:

```
CREATE TABLE EMP
    ( EMP#              CHAR(5),
      NOMEMP            CHAR(20),
      SAL               NUMERIC,
      PASATIEMPOS       SET OF ( ACTIVIDAD )
      TRABAJA PARA      COMPAÑÍA )
                  METHOD BENEFICIOS_RETIRO  ( ) : NUMERIC ;
```

Explicación: BENEFICIOS_RETIRO es un método que toma una tupia EMP dada como argumento y produce un resultado de tipo NUMERIC.

La extensión final de definición es permitir *subclases*. Por ejemplo (consulte la figura 25.3):

```
CREATE TABLE PERSONA
      ( SS#               CHAR(9),
        FECHA_NACIMIENTO DATE,
        DOMICILIO         CHAR(50)

CREATE TABLE EMP
       AS SUBCLASS OF
         PERSONA ( EMP#   CHAR(5),
         NOMEMP SAL       CHAR(20),
           PASATIEMPOS    NUMERIC,
           TRABAJA, PARA  SET OF ( ACTIVIDAD ),
METHOD                    COMPAÑÍA )
BENEFICIOS_RETIRO ( ) : NUMERIC
```

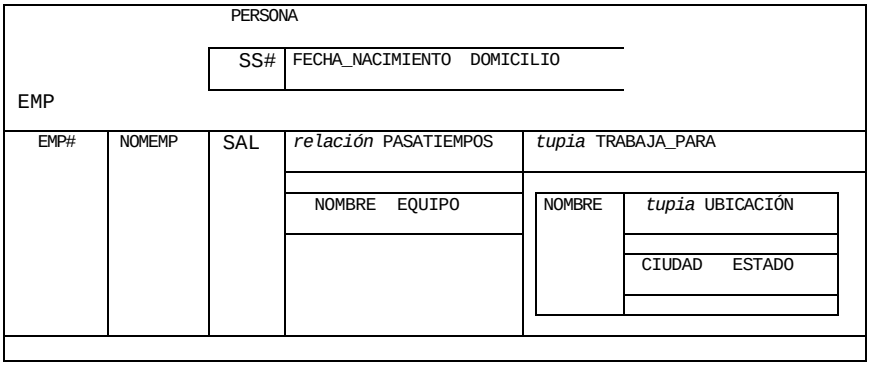


Figura 25.3 Las varrels como superclases y subclases; desaprobado.

Explicación: Ahora EMP tiene tres atributos adicionales (SS#, FECHA_NACIMIENTO y DOMICILIO) heredados de PERSONA (ya que cada ejemplar de EMP "ES UN" ejemplar de PERSONA, en términos generales). Si PERSONA tuviera algún método, también lo heredaría. *Nota:* Aquí PERSONA y EMP son ejemplos de lo que a veces se conoce como *supertablas* y *subtablas*, respectivamente. Vea la referencia [13.12] y también el apéndice B, para obtener una explicación adicional de estos conceptos junto con una crítica.

Además de las extensiones de definición que esbozamos anteriormente, también requerimos determinadas extensiones de manipulación, por supuesto; por ejemplo:

- Expresiones de ruta. Por ejemplo, EMP.TRABAJA_PARA.UBICACION.ESTADO. Ob serve que dicha expresión puede regresar escalares, tupias o relaciones en general. Observe además que en los últimos dos casos, los componentes de esas tupias o relaciones pueden ser por sí mismas, tupias o relaciones (y así sucesivamente); por ejemplo, la expresión EMP.PASATIEMPOS.NOMBRE regresa una relación. A propósito, observe que estas ex presiones de ruta *descienden* por la jerarquía de contención y en cambio, las expresiones de ruta que tratamos en el capítulo 24 van hacia *arriba.*
- Literales de tupia y de relación (posiblemente anidadas). Por ejemplo,

```
( 'E001', 'Smith', $50000,
      ( ( 'Fútbol', 11 ),     'Béisbol', 9 ) ),
                              'IBM', ( 'San José¹    ■CA' ) ) )
```

  (no pretendemos que sea la sintaxis real).
- Operadores de comparación relacional. Por ejemplo, SUBSET, SUBSETEQ y así sucesi vamente. (Los operadores particulares que mencionamos están tomados del producto par ticular bajo explicación. En ese producto, SUBSET en realidad significa "subconjunto *propio"* y SUBSETEQ significa "subconjunto"(¡!)).
- Operadores para recorrer la jerarquía de clases. *Nota:* Aquí también se necesita tener cuidado. Pudiera ser que una solicitud para recuperar la información de PERSONA junto con la información EMP asociada, produzca un resultado que no es una relación; lo que sig nifica que ha sido violada la propiedad relacional vital de *cierre,* con implicaciones potencialmente desastrosas. (Al respecto, la referencia [25.31] —la cual hace referencia a dicha clase de resultado como un "regreso accidentado"— observa simplemente sin preocupación que ¡"el programa cliente debe estar preparado para manejar la complejidad de un regreso accidentado"!)
- La capacidad de invocar métodos dentro de, por ejemplo, las cláusulas SELECT y WHERE (en términos de SQL).
- La capacidad para acceder a componentes individuales dentro de valores de atributo que sean tupias o relaciones.

Esto es suficiente para una panorámica rápida sobre la manera en que la ecuación "varrel = clase" se realiza en la práctica. Entonces, ¿qué tiene de erróneo?

Bien, observe en primer lugar que (como mencioné anteriormente) una varrel es una *variable* y una clase es un *tipo.* Entonces, ¿cómo es posible que puedan ser lo mismo? Esta primera observación debería ser lógicamente suficiente para detener de golpe la idea de que "varrel = clase". Sin embargo, hay mucho más que podemos decir y por lo tanto, acordemos suspender un poco más esta falta de creencia... Éstos son otros puntos adicionales a considerar:

La ecuación "varrel = clase" implica las siguientes ecuaciones "tupia = objeto" y "atributo = variable de ejemplar (pública)". Por lo tanto, mientras (como vimos en el capítulo 24) una clase de objetos verdadera —al menos una clase de objetos "encapsulada" o escalar— tiene métodos y no variables de ejemplar públicas, una "clase de objetos" de varrel tiene variables de ejemplar públicas y sólo de manera opcional tiene métodos (definitivamente no es "encapsulada"). Por lo tanto, de nuevo, ¿cómo es posible que las dos nociones sean lo mismo?

Hay una diferencia importante entre las definiciones de atributo (por ejemplo) "SAL NUMERIC" y "TRABAJA_PARA COMPAÑÍA". NUMERIC es un tipo de datos verdadero (equivalente a un dominio verdadero, aunque primitivo); pone una restricción, independiente del tiempo, sobre los valores que pueden aparecer legalmente en el atributo SAL. Por el contrario, COMPAÑÍA *no* es un tipo de datos verdadero; las restricciones que pone sobre los valores que pueden aparecer en el atributo TRABAJA_PARA son *dependientes del tiempo* (dependen obviamente del valor actual de la varrel COMPAÑÍA). En realidad, como señalamos anteriormente, la distinción entre varrel y dominio —o si lo prefiere usar la terminología de objetos, la distinción entre colección y clase— se ha enturbiado aquí.

Hemos visto que a los "objetos" de tupia se les permite, aparentemente, que contengan otros "objetos" de este tipo; por ejemplo, los "objetos" EMP contienen aparentemente "objetos" COMPAÑÍA. Pero en realidad no es así; en su lugar, contienen *apuntadores* hacia esos "objetos contenidos" y a los usuarios les debe quedar perfectamente claro este punto. Por ejemplo, suponga que el usuario actualiza de alguna forma una tupia específica de COMPAÑÍA (consulte nuevamente la figura 25.1). Entonces esa actualización será visible inmediatamente en todas las tupias EMP que "contienen" esa tupia de COMPAÑÍA. *Nota:* No estamos diciendo que este efecto sea indeseable, sino sólo estamos diciendo que tiene que ser explicado al usuario. Pero explicarlo al usuario equivale a decirle que el "modelo", como lo muestra la figura 25.1, es incorrecto; las tupias EMP no contienen tupias COMPAÑÍA sino que en su lugar contienen *apuntadores hacia* tupias COMPAÑÍA (como dijimos anteriormente).

Éstas son algunas implicaciones y preguntas adicionales que se desprenden de este mismo punto:

a. ¿Podemos insertar una tupia EMP y especificar un valor para la tupia COMPAÑÍA "contenida" que no exista actualmente en la varrel COMPAÑÍA? Si la respuesta es *sí,* el hecho de que el atributo TRAB AJA_PARA esté definido como si fuera de tipo COMPA ÑÍA no significa mucho, ya que no restringe significativamente la operación INSERT en forma alguna. Si la respuesta es *no,* la operación INSERT llega a ser innecesariamente compleja; el usuario tiene que especificar no sólo un nombre de compañía existente (es decir, un valor de clave externa), como se requeriría en la situación relacional equiva lente, sino también una tupia completa de COMPAÑÍA existente. Además, la especifica ción de una tupia de COMPAÑÍA completa significa, en el mejor de los casos, decirle al sistema algo que ya sabe; en el peor, significa que si el usuario comete un error, el INSERT fallará cuando podría haber tenido éxito perfectamente bien.

b. Suponga que queremos una regla ON DELETE RESTRICT para compañías (es decir, un intento de eliminar una compañía debe fallar si la compañía tiene algún empleado). Presuntamente, debemos hacer cumplir esta regla mediante código procedural, digamos mediante algún método *M* (observe que la varrel EMP no tiene una clave externa con la cual pudiera estar conectada una versión declarativa de la regla). Además, ahora no debe mos realizar las operaciones regulares de DELETE de SQL sobre la varrel COMPAÑÍA,

con excepción de las que estén dentro del código que implementa al método *M*. ¿Cómo se hace cumplir este requerimiento? Por supuesto, comentarios y preguntas similares se aplican a otras reglas de clave externa, como ON DELETE CASCADE.

c. Observe también que eliminar una tupia EMP presuntamente *no* se aplicará "en cascada" para eliminar la tupia de COMPAÑÍA correspondiente, a pesar de la pretensión de que la tupia EMP contiene esa tupia de COMPAÑÍA.

De todo lo anterior se desprende que no estamos hablando exactamente del modelo relacional. El objeto de datos fundamental ya no es una relación que contiene valores, sino que es una "relación" —de hecho, no es en absoluto una relación apropiada, en lo que se refiere al modelo relacional— que contiene valores *y apuntadores*. En otras palabras, **hemos socavado la integridad conceptual del modelo relacional.***

Suponga que definimos la vista V para que sea la proyección de EMP sobre (digamos) sólo el atributo PASATIEMPOS. Por supuesto, V también es una varrel, pero es una varrel derivada en lugar de una base. Por lo tanto, si "varrel = clase" es una ecuación correcta, V también es una clase. *¿Qué clase esl* También las clases tienen métodos, *¿cuáles métodos se aplican a V?*

Bueno, la "clase" EMP sólo tiene un método, BENEFICIOS_RETIRO, y ese método claramente no se aplica a la "clase" V. En realidad, parece no ser muy razonable que *cualquier* método que se aplique a la "clase" EMP se aplique a la "clase" V; y en realidad no hay otros. Por lo tanto, parece como si (en general) *ningún método* se aplicara al resultado de una proyección; es decir, el resultado, cualquiera que sea, en realidad no es una clase en absoluto. (Podríamos *decir* que es una clase, ¡pero esto no hace que lo sea!; tendrá variables de ejemplar públicas y no métodos, mientras que ya hemos observado que una clase "encapsulada" verdadera tiene métodos y no tiene variables de ejemplar públicas.)

En realidad es bastante claro que cuando la gente iguala las varrels y las clases, lo que tiene en mente son, específicamente, las varrels base y se olvida de las derivadas. (En realidad, los apuntadores que tratamos anteriormente son apuntadores hacia tupias en varrels base, no en derivadas.) Pero distinguir de esta forma entre varrels base y derivadas es un error muy grande, debido a que la cuestión de cuáles varrels son base y cuáles derivadas es arbitraria en un sentido muy importante (recuerde la explicación del *principio de intercambiabilidad* del capítulo 9).

Por último, *¿qué dominios están soportados!* Aquellos partidarios de la ecuación "varrel = clase" nunca parecen tener mucho que decir acerca de los dominios, presuntamente debido a que no pueden ver la manera en que los dominios, como tales, encajan en su esquema general. Y además, es claro que los dominios son esenciales, como sabemos a partir de muchas de nuestras primeras discusiones (vea, por ejemplo, el capítulo 3).

---

* El término *integridad conceptual* se debe a Fred Brooks, que al respecto dice lo siguiente [25.1]: "La integridad [conceptual] es *la* consideración más importante en el diseño de sistemas. Es mejor hacer que un sistema omita determinadas características anómalas [y] refleje un conjunto de ideas de diseño, en lugar de tener uno que contenga muchas ideas buenas pero independientes y no coordinadas" (cursivas en el original). Y 20 años después, añade: "Un producto de programación limpio y elegante debe presentar... un modelo mental coherente... La integridad [conceptual]... es el factor más importante en la facilidad de uso... **Actualmente estoy más convencido que nunca.** La integridad conceptual es central para la calidad del producto" (negritas y cursivas en el original).

El mensaje general de lo anterior puede ser resumido de la siguiente forma. Obviamente es posible construir sistemas que estén basados en la ecuación equivocada "varrel = clase"; de hecho, ya existen algunos de éstos. Es igualmente obvio que esos sistemas (al igual que un automóvil sin aceite en su motor o una casa construida sobre arena) pueden incluso proporcionar un servicio útil por un tiempo, pero están condenados a una falla eventual.

¿De dónde vino el primer gran error garrafal?

Es interesante especular sobre el origen del primer gran error garrafal. Nos parece que tiene sus raíces en la falta de consenso (que mencionamos en el capítulo 24) sobre el significado de los términos en el mundo de los objetos. En particular, el propio término *objeto* no tiene un significado aceptado universalmente y con el que todos estén de acuerdo; que es la razón por la cual preferimos no usarlo mucho.

No obstante los comentarios anteriores, queda bastante claro que en los círculos de los lenguajes de programación de objetos —en cualquier nivel— el término *objeto* se refiere a lo que más tradicionalmente sería llamado un *valor* o una *variable* (o posiblemente ambos). Sin embargo, por desgracia el término también es usado en otros círculos; en particular, es usado en determinados círculos de *modelado semántico* como parte de las diversas técnicas y metodologías de "análisis y diseño de objetos" o "modelado de objetos" (vea, por ejemplo, la referencia [13.3]). Y en esos círculos parece claro que el término *no* significa un valor o una variable, sino lo que la comunidad de base de datos podría llamar más generalmente una *entidad* (lo que implica [a propósito] entre otras cosas que, a diferencia de los objetos de un lenguaje de programación, dichos objetos en definitiva no están encapsulados). En otras palabras, "el modelado de objetos" es en realidad simplemente el "modelado de entidad-vínculo" con otro nombre; de hecho, la referencia [13.3] lo admite en cierta medida. Por consecuencia, las cosas que en tales metodologías son identificadas como "objetos", después son transformadas (correctamente) hacia tupias en varrels, en lugar de valores en dominios. ¡ Abracadabra!

25.3 EL SEGUNDO GRAN ERROR GARRAFAL

En esta sección analizamos **el segundo gran error garrafal;** como veremos, este segundo error garrafal parece ser consecuencia lógica del primero, pero también es importante por derecho propio (además, también se le puede cometer por derecho propio, aun cuando el primer gran error garrafal sea evitado). El error garrafal consiste en **la mezcla de apuntadores** y **relaciones.** Comenzamos revisando las características principales del enfoque "varrel = clase", tal como lo identificamos en la sección anterior. Es posible que algunas personas hayan encontrado esa sección un poco confusa, debido a que algunas de las características que aparentemente estábamos objetando eran características sobre las cuales habríamos argumentado a su favor en partes anteriores del libro (un caso podrían ser los atributos con valor de tupia y de relación). Por lo tanto:

- *Atributos con valor de tupia y de relación.* De hecho, no objetamos tales atributos (¿cómo podríamos?). Lo que objetamos es (a) la idea de que tales atributos deban tener muy específicamente valores que aparecen actualmente en alguna otra varrel (base) y (b) la idea de que tales atributos en realidad tienen valores que no son tupias o relaciones en sí (sino que en su lugar, son **apuntadores** hacia tupias o relaciones); lo cual significa, por supuesto,

que en realidad no estamos hablando de atributos con valor de tupia o de relación en forma alguna. *Nota:* En realidad, la idea de apuntadores que apuntan hacia "tupias o relaciones" —lo que significa específicamente a *valores* de tupia o de relación— no tiene sentido alguno. Dentro de unos momentos trataremos este punto en detalle.

- *La asociación de operadores ("métodos") con varrels.* Tampoco tenemos objeciones para esta idea; es básicamente la noción de procedimientos almacenados o disparados, bajo otro nombre. Pero lo que objetamos es la idea de que tales operadores deban estar asociados con varrels (y sólo varrels) y no con dominios o tipos. También objetamos la idea de que deban estar asociados con *una* varrel *específica* (la noción de operandos de destino bajo otro nombre).
- *Subclases y superclases.* Aquí sí objetamos... En un sistema que iguala varrels y clases, las subclases y las superclases se convierten en *subtablas* y *supertablas,* y esta noción es una con la que somos muy escépticos [13.12]. Queremos soporte de herencia adecuado como describimos en el capítulo 19.
- *Expresiones de ruta.* No tendríamos objeciones para las expresiones de ruta que fueran sim plemente abreviaturas sintácticas para el seguimiento de determinadas referencias asocia tivas; por ejemplo, de una clave externa a una clave candidata coincidente, como se propone en la referencia [25.11]. Sin embargo, las expresiones de ruta, tal como las tratamos en la sección 25.2 son, en su lugar, abreviaturas para determinadas *cadenas de apuntadores* y esto sí lo objetamos (debido a que, en primer lugar, objetamos los apuntadores).
- *Literales de tupia y de relación.* Éstas son esenciales, aunque necesitan ser generalizadas hacia *selectores* de tupia y de relación [3.3].
- *Operadores de comparación relaciónales.* También son esenciales (aunque deben hacerse en la forma correcta).
- *Operadores para el recorrido de la jerarquía de clases.* Si "jerarquía de clases" significa en realidad "jerarquía de varrels", entonces (como indicamos en la sección anterior) tene mos grandes objeciones, debido a la probable violación de la propiedad de cierre relacional (vea por ejemplo la referencia [25.31]). Si significara "jerarquía de tipos" en el sentido del capítulo 19, entonces no tendríamos objeciones (pero no es así).
- *Invocación de métodos en, por ejemplo, cláusulas SELECT y WHERE.* Por supuesto.
- *Acceso a componentes individuales dentro de valores de atributo que son tupias o relacio nes.* Por supuesto.

Enfoquémonos ahora en el asunto de mezclar apuntadores y relaciones. El punto principal del argumento es muy simple. Por definición, los apuntadores apuntan hacia *variables* y no hacia *valores* (debido a que las variables tienen direcciones y los valores no). Por lo tanto, por definición, si permitimos que la varrel Rl tenga un atributo cuyos valores son apuntadores "hacia" la varrel R2, entonces esos apuntadores apuntan hacia *variables* de tupia y no hacia *valores* de tupia. **Pero en el modelo relacional no hay algo que equivalga a una variable de tupia.** El modelo relacional maneja valores de relación, los cuales son (en términos generales) conjuntos de valores de tupia, quienes a su vez (de nuevo, en términos generales), son conjuntos de valores escalares. También maneja variables de relación que son variables cuyos valores son relaciones. Sin embargo, *no* maneja variables de tupia (que son variables cuyos valores son tupias) o variables escalares (que son variables cuyos valores son escalares). El *único* tipo de variables incluido en el modelo relacional (y el único tipo de variables permitido en una base de datos relacional)

es, muy específicamente, la variable de *relación. De esto se desprende que la idea de mezclar apuntadores y relaciones constituye una desviación MA YOR del modelo relacional, al introducir un tipo de variable completamente nuevo.* Como indicamos en la sección anterior, podemos de hecho argumentar que socava seriamente la integridad conceptual del modelo relacional.

En las referencias [24.21] y [25.11] puede encontrar argumentos más detallados que apoyan esta posición. Vea también las referencias [25.8] a [25.10] y [25.13], las cuales exponen la noción relacionada —e importante— de *esencialidad* (de las construcciones de datos dentro de un modelo de datos).

Por lo tanto, dado el argumento anterior, es lastimoso ver que la mayoría (y posiblemente) todos los productos de objetos/relaciónales de la cosecha actual —incluso aquellos que evitan el primer gran error garrafal— parecen, no obstante, estar mezclando apuntadores y relaciones exactamente en la manera que acabamos de explicar y objetar anteriormente. Cuando Codd definió por primera vez el modelo relacional excluyó deliberadamente los apuntadores. Para citar la referencia [5.2]:

> Es seguro suponer que todos los tipos de usuario [incluyendo a los usuarios finales, en particular] comprenden el acto de comparar valores, pero relativamente pocos comprenden las complejidades de los apuntadores. El modelo relacional está basado en este principio fundamental... [La] manipulación de apuntadores es más propensa a errores que el acto de comparar valores, aunque el usuario comprenda las complejidades de los apuntadores.

Para ser específicos, los apuntadores conducen a la persecución de apuntadores y la persecución de apuntadores es notoriamente propensa a errores. Como indicamos en el capítulo 24, es precisamente este aspecto de los sistemas de objetos el que da lugar a las críticas, que se oyen a veces, de que tales sistemas "se parecen al CODASYL recalentado".

¿De dónde vino el segundo error garrafal?

Es difícil encontrar una justificación real en la literatura para el segundo gran error garrafal (es decir, cualquier justificación técnica; pero hay evidencia de que la justificación no es técnica sino política). Por supuesto, dado el hecho de que los sistemas y lenguajes de objetos incluyen apuntadores (en forma de IDs de objetos), la idea de mezclar apuntadores y relaciones se presenta casi ciertamente a partir de un deseo de hacer que los sistemas relaciónales sean más "parecidos a objetos"; aunque esta "justificación" sólo empuja el problema hacia otro nivel. Ya hemos dejado perfectamente claro que (en nuestra opinión) los sistemas de objetos exponen a los apuntadores ante los usuarios precisamente debido a que fallan en distinguir adecuadamente entre modelo y su implementación.

Por lo tanto, sólo podemos conjeturar que la razón por la cual la idea de mezclar apuntadores y relaciones esté siendo tan ampliamente promulgada se debe, en primer lugar, a que muy pocos comprenden por qué los apuntadores fueron excluidos de las relaciones. Como dijo Santayana: *quienes no pueden recordar el pasado están condenados a repetirlo* (citado usualmente en la forma "quienes no saben la historia están condenados a repetirla"). En estos asuntos estamos completamente de acuerdo con Maurice Wilkes cuando escribe [25.35]:

> Quisiera ver que pusieran la enseñanza de la ciencia de la computación deliberadamente en un marco de referencia histórico... Los estudiantes necesitan comprender por qué se ha llegado a la situación actual, lo que se ha intentado, lo que ha funcionado y lo que no, y cómo las mejoras en hardware hacen posible el progreso. La ausencia de este elemento en su capacitación hace que la gente se aproxime desde sus inicios a cada uno de los problemas. Son

capaces de proponer soluciones que han sido deficientes en el pasado. En vez de andar sobre los hombros de sus precursores, tratan de recorrer el camino solos.

25.4 CUESTIONES DE IMPLEMENT ACIÓN

Una implicación importante del soporte apropiado para los tipos de datos es que permite que los fabricantes de otras compañías (así como los mismos fabricantes de DBMS) construyan y vendan paquetes de "tipos de datos" separados que pueden en efecto ser incorporados al DBMS. Los ejemplos incluyen paquetes para soportar el manejo de texto sofisticado, el procesamiento de series de tiempo financieras, el análisis de datos geoespaciales (cartográficos), etcétera. A dichos paquetes se les conoce de diversas formas: "navajas de datos" (Informix), "cartuchos de datos" (Oracle), "extensores relaciónales"* (IBM), etcétera. En lo que viene a continuación nos mantendremos con el término *paquetes de tipos.*

Sin embargo, la incorporación de un nuevo paquete de tipos al sistema es una labor no trivial, y la capacidad de hacerlo tiene implicaciones importantes para el diseño y la estructura del propio DBMS. Para ver la razón de ello en ambos casos, considere lo que pasaría (por ejemplo) si alguna consulta incluyera referencias hacia datos de algún tipo definido por el usuario o invocaciones de algún operador definido por el usuario (o ambos):

- En primer lugar, el compilador del lenguaje de consulta tiene que ser capaz de analizar sin tácticamente y verificar el tipo que se solicita, por lo que tiene que saber algo acerca de esos tipos y operadores definidos por el usuario.
- Segundo, el optimizador tiene que ser capaz de decidir un plan de consulta adecuado para esa solicitud y por lo tanto, también debe estar consciente de determinadas propiedades de esos tipos y operadores definidos por el usuario. En particular, debe saber la manera en que están almacenados físicamente los datos (vea el siguiente párrafo).
- Tercero, el componente que administra el almacenamiento físico tiene que soportar las es tructuras de almacenamiento más recientes ("quadtrees, árboles-R, etcétera") que men cionamos en nuestra explicación del problema de los rectángulos en la sección 25.1. Puede incluso tener que soportar la posibilidad de que los usuarios con los conocimientos ade cuados, introduzcan nuevas estructuras de almacenamiento y métodos de acceso propios [25.21], [25.33].

El efecto neto de lo anterior es que el sistema necesita ser *extensible;* en realidad, extensible en varios niveles. Trataremos brevemente cada nivel.

Análisis sintáctico y verificación de tipos

En un sistema convencional, la información referente a éstos puede estar (y por lo general está) "integrado" en el compilador del lenguaje de consulta, ya que los únicos tipos de operadores disponibles están todos integrados. Por el contrario, en un sistema en el cual los usuarios pueden definir sus propios tipos y operadores, es claro que este enfoque "integrado" no va a funcionar. Por lo tanto, en vez de ello, lo que va a suceder es esto:

---

* Un término *excesivamente* inadecuado, en nuestra opinión.

1. La información con respecto a los tipos y operadores definidos por el usuario —y posible mente también sobre los tipos y operadores integrados— se mantiene en el catálogo del sis tema. Este hecho implica que el propio catálogo necesita ser rediseñado (o al menos extendido); también implica que la introducción de un nuevo paquete de tipos involucra mucha actualización del catálogo. (Por supuesto, en términos del **Tutorial D,** esa actuali zación es realizada en segundo plano como parte del proceso de ejecución de las declara ciones de definición TYPE y OPERATOR aplicables.)
2. El compilador mismo necesita ser reescrito para que acceda al catálogo a fin de obtener la in formación necesaria de tipos y operadores. Luego puede usar esa información para realizar toda la verificación de tipos en tiempo de compilación, como vimos en los capítulos 5, 8 y 19.

Optimización

Aquí hay muchos aspectos involucrados, y en este libro sólo podemos rasgar la superficie del problema. Sin embargo, al menos podemos señalar cuáles son algunos de éstos:

- *Transformación de expresiones ("reescritura de consultas ").* Un optimizador convencional aplica determinadas leyes de transformación para reescribir las consultas, como vimos en el capítulo 17. Sin embargo, esas leyes de transformación han estado históricamente "in tegradas" en el optimizador (debido nuevamente a que los tipos de datos y operadores disponibles han sido los integrados). Por el contrario, en un sistema de objetos/relacional el conocimiento relevante (al menos, lo que se aplica específicamente a los tipos y opera dores definidos por el usuario) necesita ser mantenido en el catálogo; lo que implica más extensiones al catálogo y también que el optimizador mismo deba ser reescrito. Éstas son algunos ejemplos:
  a. Dada una expresión como NOT (CANT > 500), un buen optimizador convencional la trans formará hacia CANT < 500 (debido a que la segunda versión puede utilizar un índice sobre CANT mientras que la primera no puede). Por razones similares debe haber una forma para informar al optimizador cuando un operador definido por el usuario es la negación de otro.
  b. Un buen optimizador convencional también sabrá que, por ejemplo, las expresiones CANT > 500 y 500 < CANT son lógicamente equivalentes. Debe haber una forma para informar al optimizador cuando dos operadores definidos por el usuario son opuestos en este sentido.
  c. Un buen optimizador convencional también sabrá que, por ejemplo, los operadores "+" y "-" se cancelan (es decir, son inversos); por ejemplo, la expresión CANT + 500 - 500 se reduce simplemente a CANT. Debe haber una forma para informar al optimizador cuando dos operadores definidos por el usuario son inversos en este sentido.
- *Selectividad.* Dada una expresión lógica, como CANT > 500, los optimizadores hacen generalmente especulaciones sobre la *selectividad* de esa expresión (es decir, el porcentaje de tupias que la hacen *verdadera).* Para los tipos de datos y operadores integrados, de nuevo, esa información de selectividad puede estar "integrada" en el optimizador; por el contrario, para los tipos y operadores definidos por el usuario, necesita haber una forma de proporcionar algún código al optimizador, definido por el usuario, que pueda ser llamado para especular sobre las selectividades.
- *Fórmulas de costo.* El optimizador necesita saber cuánto cuesta ejecutar un operador definido por el usuario. Tomando una expresión como *p* AND *q,* por ejemplo, donde *p* es

(digamos) una invocación al operador AREA sobre algún polígono complicado y *q* es una comparación simple como CANT > 500, probablemente preferiríamos que el sistema ejecutara primero a *q*, a fin de que *p* se ejecute sólo sobre las tupias para las cuales *q* da como resultado *verdadero*. De hecho, algunas de las técnicas clásicas de transformación de expresiones, tal como hacer siempre las restricciones antes que las juntas, no son necesariamente válidas para los tipos y operadores definidos por el usuario [25.7], [25.18].

- *Estructuras de almacenamientos y métodos de acceso.* El optimizador debe estar claramente consciente de las estructuras de almacenamiento y los métodos de acceso que se efectúan (vea la siguiente subsección).

Estructuras de almacenamiento

Debería ser obvio que los sistemas objeto-relacionales van a necesitar más formas —posiblemente muchas más— de almacenamiento y acceso a datos en el nivel físico, que las que han proporcionado tradicionalmente (por ejemplo) los sistemas SQL. Éstas son algunas consideraciones importantes:

- *Nuevas estructuras de almacenamiento.* Como ya indicamos, el sistema probablemente necesitará soportar nuevas estructuras de almacenamiento "integradas" (árboles-R, etcétera) y tal vez también necesitará una forma para que los usuarios que tengan las habilidades ade cuadas introduzcan estructuras de almacenamiento y métodos propios de acceso adicionales.
- *índices sobre datos de un tipo definido por el usuario.* Los índices tradicionales están basa dos en datos de algún tipo integrado y un entendimiento integrado de lo que significa el ope rador "<". En un sistema objeto-relacional debe ser posible construir índices sobre los datos de un tipo definido por el usuario, basado en la semántica del operador aplicable "<" definido por el usuario (por supuesto, dando por hecho en primer lugar, que dicho operador haya sido definido).
- *Índices sobre resultados de un operador.* Es probable que no tenga mucho sentido construir un índice directamente sobre un conjunto de valores de tipo POLÍGONO; lo más probable es que todo lo que haría este índice sería ordenar los polígonos de acuerdo con su codificación interna de la cadena de bytes.* Sin embargo, un índice basado en las *áreas* de estos polígonos podría ser muy útil. *Nota:* En el capítulo 21 nos referimos a estos índices como índices *funcionales*.

25.5 BENEFICIOS DE UN *ACERCAMIENTO* VERDADERO

En la referencia [25.31], Stonebraker presenta una "matriz de clasificación" para los DBMSs (vea la figura 25.4). El **cuadrante 1** de esa matriz representa las aplicaciones que sólo manejan datos muy simples y no tienen requerimientos de consultas *ad hoc* (un procesador texto tradicional es un buen ejemplo). En realidad, dichas aplicaciones no son aplicaciones de bases de datos en el sentido clásico del término; el "DBMS" que mejor satisface sus necesidades es, simplemente, el sistema de archivos proporcionado como parte del sistema operativo subyacente.

---

* Recuerde del capítulo 24 que los sistemas proporcionan tradicionalmente un tipo de datos BLOB para manejar "objetos binarios grandes". En un sistema de objetos/relacional, los valores de datos de ciertos tipos definidos por el usuario bien pudieran ser guardados físicamente como BLOBs.

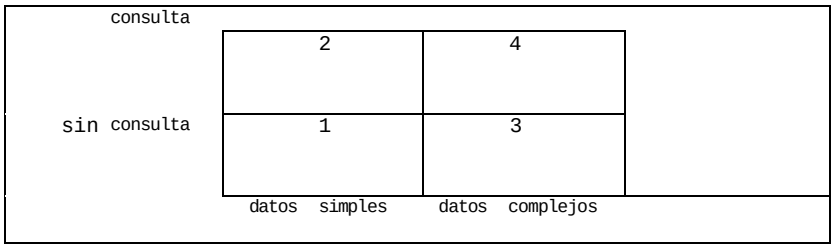


Figura 25.4 Matriz de clasificación de DBMS de Stonebraker.

El **cuadrante 2** representa aplicaciones que tienen requerimientos de consultas *ad hoc,* pero que todavía manejan sólo datos muy simples. La mayoría de las aplicaciones de negocios actuales caen en este cuadrante, y están bastante bien soportadas por los DBMSs relaciónales tradicionales (o al menos SQL).

El **cuadrante 3** representa aplicaciones con datos y requerimientos de procesamiento complejos pero sin requerimientos de consultas *ad hoc.* Por ejemplo, las aplicaciones CAD/CAM pueden caer en este cuadrante. Los DBMSs de objetos actuales están orientados principalmente a este segmento del mercado (por lo general, los productos SQL tradicionales no hacen un buen trabajo con las aplicaciones del cuadrante 3).

Por último, el **cuadrante 4** representa aplicaciones que necesitan datos complejos y consultas *ad hoc* contra los datos. Stonebraker da un ejemplo de una base de datos que contiene fotografías digitalizadas de 35 mm, donde una consulta típica es "obtener las fotografías de atardeceres tomadas en las 20 millas alrededor de Sacramento, California". Luego continúa dando argumentos para soportar su posición de que (a) se requiere un DBMS de objetos/relacional para aplicaciones que caen en este cuadrante y (b) durante los próximos años la mayoría de las aplicaciones caerá o se moverá hacia este cuadrante. Por ejemplo, incluso una simple aplicación de recursos humanos podría expandirse para incluir fotografías de los empleados, grabaciones de sonido (mensajes hablados) y cosas similares.

En suma, Stonebraker está argumentando (y estamos de acuerdo con 61) que los "sistemas de objetos/relaciónales están en el futuro de todos"; no son simplemente una novedad pasajera que pronto será reemplazada por alguna otra idea brevemente atractiva. Sin embargo, tal vez debamos recordar que, por lo que a nosotros concierne, un sistema de objetos/relacional verdadero es, simplemente, un sistema *relacional* verdadero. En particular, ¡es un sistema que no comete ninguno de los dos grandes errores garrafales! Stonebraker no parece estar de acuerdo con nuestra posición; al menos, en la referencia [25.31] nunca dice mucho de esto, y en realidad ésta implica que la mezcla de apuntadores y relaciones no sólo es aceptable sino que es deseable (y de hecho, necesaria).

Sea como fuere, podríamos argumentar que un sistema de objetos/relacional *genuino* resolvería todos los problemas que (como dijimos en el capítulo anterior) son realmente problemas de los sistemas que son sólo sistemas de objetos llanos, no de objetos/relaciónales. Para ser específicos, dicho sistema deberá ser capaz de soportar todo lo siguiente sin gran dificultad:

- Consultas *ad hoc,* definiciones de vistas y restricciones de integridad declarativas;
- Métodos que abarcan clases (no hay necesidad de distinguir un operando de "destino");
- Clases definidas dinámicamente (para los resultados de las consultas *ad hoc);*

- Acceso en modo dual (no enfatizamos el punto en el capítulo 24, pero los sistemas de ob jetos no soportan, por lo general, el principio de modo dual; en su lugar, usan lenguajes diferentes para el acceso programado e interactivo hacia la base de datos);
- Verificación de integridad diferida (en tiempo del COMMIT);
- Restricciones de transición;
- Optimization semántica;
- Vínculos de grado mayor a dos;
- Reglas de clave externa (ON DELETE CASCADE, etcétera);
- Optimizabilidad;

etcétera. Además:

- Los OIDs y la caza de apuntadores están ahora totalmente "protegidos" y ocultos ante el usuario;
- Desaparecen las cuestiones de objetos "difíciles" (por ejemplo, ¿qué significa la junta de dos objetos?);
- Los beneficios de la encapsulación (tal como son) todavía se aplican, pero a valores es calares dentro de relaciones, no a las relaciones en sí;
- Los sistemas relaciónales pueden manejar ahora áreas de aplicaciones "complejas" tales como CAD/CAM (como explicamos anteriormente).

Y el enfoque queda conceptualmente claro.

25.6 RESUMEN

Hemos examinado brevemente el campo de los sistemas **de objetos/relaciónales.** Tales sistemas sólo son (o deberían ser) sistemas *relaciónales* que soportan adecuadamente el concepto de **dominio** relational (es decir, tipos); lo que significa, en particular, que los usuarios son (o deberían ser) capaces de definir sus propios tipos. No necesitamos hacer nada al modelo relacional para lograr la funcionalidad de objetos que deseamos (salvo implementarla).

Luego examinamos los dos **grandes errores garrafales.** El primero es igualar las clases de objetos y las varrels (una ecuación que, desgraciadamente, es demasiado atractiva, al menos en la superficie). Especulamos que los errores garrafales se presentan por una confusión sobre dos interpretaciones bastante distintas del término *objeto.* Explicamos un ejemplo (en detalle) que muestra cómo podría lucir un sistema que comete el primer gran error garrafal y explicamos algunas de las consecuencias de ese error. Una consecuencia principal es que ¡parece conducir directamente a cometer también el segundo gran error garrafal; es decir, mezclar apuntadores y relaciones (aunque, en realidad, este segundo error garrafal puede cometerse sin el primero, y casi todos los sistemas que hay en el mercado, por desgracia, parece que lo cometen). Nuestra posición es que el segundo gran error garrafal **socava la integridad conceptual del modelo relacional** en muchas formas. En particular, viola el *principio de intercambiabilidad* de relaciones base y derivadas.

Después, examinamos brevemente algunas cuestiones de implementación. El punto primordial es que la incorporación de un nuevo "paquete de tipos" afecta por lo menos a los com-

ponentes del compilador, optimizador y administrador de almacenamiento del sistema. Por consecuencia, un sistema de objetos/relacional no puede ser implementado —al menos, no adecuadamente— imponiendo simplemente una nueva capa de código sobre un sistema relacional existente; más bien, el sistema debe ser reconstruido desde sus fundamentos para hacer extensible a cada componente individual conforme sea necesario.

Por último, vimos la *matriz de clasificación de DBMS* de Stonebraker y tratamos brevemente los beneficios que pueden surgir de un *acercamiento* verdadero entre las tecnologías de objetos y la relacional (donde por "verdadero" queremos decir, en particular, que el sistema en cuestión no comete ninguno de los dos grandes errores garrafales).

REFERENCIAS Y BIBLIOGRAFÍA

Han sido construidos varios prototipos de objetos/relaciónales en los últimos años. Dos de los más conocidos, y que más han influido, son **Postgres** de la Universidad de California en Berkeley [25.26], [25.30], [25.32] y **Starburst** de IBM Research [25.14], [25.17], [25.21] y [25.22]. Hacemos notar que (al menos en su forma original) ninguno de estos sistemas se adhirió a la ecuación "obviamente correcta" *dominio = clase*.

También debemos mencionar que SQL3 incluye varias características que están dirigidas específicamente para soportar los sistemas de objetos/relaciónales (vea el apéndice B).

25.1 Frederick P. Brooks, Jr.: *The Mythical Man-Month* (edición del 20 aniversario). Reading, Mass.: Addison-Wesley (1995).

25.2 Michael J. Carey, Nelson M. Mattos y Anil K. Nori: "Object/Relational Database Systems: Prin ciples, Products, and Challenges", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data. Tucson, Ariz, (mayo, 1997).

Para citar: "Los tipos de datos abstractos, las funciones definidas por el usuario, los tipos de filas, las referencias, la herencia, las subtablas, las colecciones, los disparadores, ¿qué son?" ¡Una buena pregunta! Hay ocho características en la lista (y hay una suposición tácita de que todas son características específicas de SQL3). De estas ocho podríamos argumentar que al menos cuatro son indeseables, otras dos son parte de lo mismo y las otras dos son ortogonales con respecto a la pregunta de si el sistema es o no de objetos/relacional. Vea el apéndice B.

25.3 Michael J. Carey *et al:* "The BUCKY Object/Relational Benchmark", Proc. 1997 ACM SIG MOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Del resumen: "BUCKY (Benchmark of Universal or Complex Kwery Ynterfaces *[sic])* es una evaluación comparativa orientada a consultas que prueba muchas de las características principales proporcionadas por los sistemas de objetos/relaciónales, incluyendo tipos de filas y herencia, referencias y expresiones de ruta, conjuntos de valores atómicos y de referencias, métodos y enlace tardío, y tipos de datos abstractos definidos por el usuario y sus métodos".

25.4 R. G. G. Cattell: "What Are Next-Generation DB Systems?", *CACM 34,* No. 10 (octubre, 1991).

25.5 Donald D. Chamberlin: "Relations and References—Another Point of View", *InfoDB 10,* No. 6 (abril, 1997).

Vea el comentario a la referencia [25.11]

25.6 Surajit Chaudhuri y Luis Gravano: "Optimizing Queries over Multi-Media Repositories", Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada (junio, 1996).

Las bases de datos de objetos/relaciónales bien pueden ser usadas como "depósitos multimedia". La consulta contra los datos multimedia, no sólo produce generalmente un conjunto de objetos resultantes, sino también un *grado de coincidencia* para cada uno de estos objetos que indica qué tan bien satisfacen la condición de búsqueda (por ejemplo, "el grado de color rojo" de una imagen). Tales consultas pueden especificar un umbral sobre el grado de coincidencia y también pueden especificar una *cuota* [6.4]. Este artículo considera la optimización de dichas consultas.

25.7 Surajit Chaudhuri y Kyuseok Shim: "Optimization of Queries with User-Defined Predicates", Proc. 22nd Int. Conf. on Very Large Data Bases, Mumbai (Bombay), India (septiembre, 1996).

25.8 E. F. Codd y C. J. Date: "Interactive Support for Nonprogrammers: The Relational and Net work Approaches", en C. J. Date, *Relational Database: Selected Writings.* Reading, Mass.: Addison-Wesley (1986).

Es el artículo que presenta la noción de *esencialidad,* un concepto que es crucial para la comprensión adecuada de los modelos de datos (¡en ambos sentidos del término!; vea el capítulo 1, sección 1.3). El modelo relacional tiene básicamente sólo una construcción de datos esencial, la relación misma. Por el contrario, el modelo de objetos tiene muchas: conjuntos, bolsas, listas, arreglos, etcétera (sin mencionar los IDs de objetos). Para una mayor explicación, vea las referencias [25.9], [25.10] y [25.13].

25.9 C. J. Date: "Support for the Conceptual Schema: The Relational and Network approaches", en *Relational Database Writings 1985-1989.* Reading, Mass.: Addison-Wesley (1990).

Un argumento en contra de la mezcla de apuntadores y relaciones [25.11] es la complejidad que causan los apuntadores. Este artículo incluye un ejemplo que ilustra muy claramente el punto (vea las figuras 25.5 y 25.6 más adelante).

25.10 C. J. Date: "Essentiality", en *Relational Database Writings 1991-1994.* Reading, Mass.: Addison-Wesley (1995).

25.11 C. J. Date: "Don't Mix Pointers and Relations!" y "Don't Mix Pointers and Relations—*Please!",* ambos en C. J. Date, Hugh Darwen y David McGoveran, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).

El primero de estos dos artículos argumenta fuertemente en contra del segundo gran error garrafal. En la referencia [25.5], Chamberlin proporciona una refutación a algunos de los argumentos de este primer artículo. El segundo artículo fue escrito como respuesta directa a la refutación de Chamberlin.

25.12 C. **J.** Date: "Objects and Relations: Forty-Seven Points of Light", en C. J. Date, Hugh Darwen y David McGoveran, *Relational Database Writings 1994-1997.* Reading, Mass.: Addison-Wesley (1998).

Una respuesta mano a mano a la referencia [25.19].

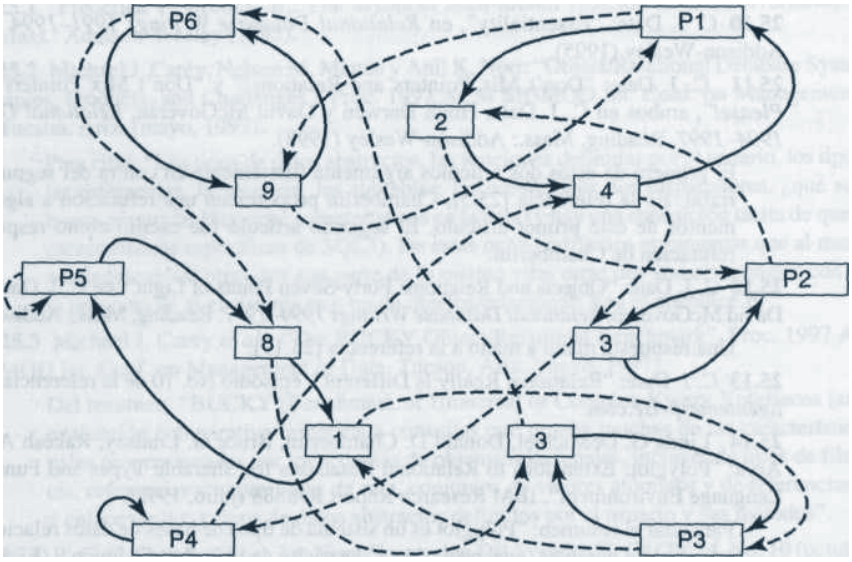
25.13 C. **J.** Date: "Relational Really Is Different", episodio No. 10 de la referencia [5.9]. *www.intelligententerprise. com.*

25.14 Linda G. DeMichiel, Donald D. Chamberlin, Bruce G. Lindsay, Rakesh Agrawal y Manish Arya: "Polyglot: Extensions to Relational Databases for Sharable Types and Functions in a Multi-Language Environment", IBM Research Report RJ8888 (julio, 1992).

Para citar el resumen: "Polyglot es un sistema de tipos de bases de datos relaciónales extensibles que soporta herencia, encapsulación y despacho de métodos dinámico". (El *despacho de métodos dinámico* es otro término para el enlace en tiempo de ejecución.) Para continuar: "[Polyglot] permite el uso de varios lenguajes de aplicación y permite que los objetos conserven su comportamiento cuando cruzan la frontera entre la base de datos y el programa de aplicación. Este artículo describe el diseño de Polyglot, las extensiones al lenguaje SQL para soportar el uso de los tipos y métodos de Polyglot y la implementation de Polyglot en el [prototipo] relacional Starburst".

P#_MAYOR	P#_MENOR	CANT
P1	P2	2
P1	P4	4
P5	P3	1
P3	P6	3
P6	P1	9
P5	P6	8
P2	P4	3

Figura 25.5 Una relación de lista de materiales.



Flechas continuas: "lista de materiales"
Flechas punteadas: "donde se usan"

Figura 25.6 Analogía de la figura 25.5 basada en apuntadores.

Polyglot está atacando claramente el tipo de cuestiones que son el tema del presente capítulo (así como de los capítulos 5, 19 y 24). Sin embargo, son importantes algunos comentarios. Primero, el término relacional **dominio** (de manera sorprendente) nunca es mencionado. Segundo, Polyglot proporciona los generadores de tipos integrados (el término de Polyglot es *metatipos)* **tipo base, tipo tupia, tipo renombrar, tipo arreglo** y **tipo lenguaje,** pero (otra vez, de manera sorprendente) no el **tipo relación.** Sin embargo, el sistema está diseñado para permitir la introducción de generadores de tipos adicionales.

25.15 David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel y Jie-Bing Yu: "Client-Server Para dise", Proc. 20th Int. Conf. on Very Large Data Bases, Santiago, Chile (septiembre, 1994).

Paradise —"Sistema de Información de Datos en Paralelo"— es un prototipo de objetos/relacional (originalmente "relacional extendido") de la Universidad de Wisconsin, "diseñado para el manejo de aplicaciones GIS" (GIS = sistema de información geográfico). Este artículo describe el diseño y la implementación de Paradise.

25.16 Michael Godfrey, Tobias Mayr, Praveen Seshadri y Thorsten von Eichen: "Secure and Portable Database Extensibility", Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash, (junio, 1998).

"Debido a que los operadores definidos por el usuario son proporcionados por clientes desconocidos o no confiables, el DBMS debe tener cuidado de operadores que puedan hacer fallar al sistema, o modificar directamente sus archivos o la memoria (haciendo a un lado los mecanismos de autorización), o monopolizar los recursos de la CPU, la memoria o el disco" (ligeramente cambiado). Obviamente se necesitan controles. Este artículo es un reporte sobre investigaciones sobre este tema usando Java y el prototipo de objetos/relacional PREDATOR [25.24]. Concluye, en forma motivadora, que un sistema de base de datos "puede soportar extensibilidad segura y transportable, usando Java sin sacrificar excesivamente el rendimiento".

25.17 Laura M. Haas, J. C. Freytag, G. M. Lohman y Hamid Pirahesh: "Extensible Query Processing in Starburst", Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (junio, 1989).

El propósito del proyecto Starburst se expandió, en cierta forma, después de que fue escrito el artículo original [25.21]: "Starburst proporciona soporte para la incorporación de nuevos métodos de almacenamiento para tablas; nuevos tipos de métodos de acceso y restricciones de integridad; nuevos tipos de datos; funciones, y nuevas operaciones de tablas". El sistema está dividido en dos componentes principales Core y Corona, que corresponden respectivamente a RSS y RDS en el prototipo original del System R (vea la referencia [4.2] para una explicación de estos dos componentes del System R). Core soporta las funciones de extensibilidad descritas en la referencia [25.21]. Corona soporta el lenguaje de consulta Hydrogen de Starburst, que es un dialecto de SQL que (a) elimina la mayoría de las restricciones de implementación del SQL del System R, (b) es mucho más ortogonal que el SQL del System R, (c) soporta consultas recursivas y (d) es extensible por el usuario. Este artículo incluye una exposición interesante sobre la "reescritura de consultas"; es decir, las reglas de transformación de expresiones (vea el capítulo 17). Vea también la referencia [17.50].

25.18 Joseph M. Hellerstein y Jeffrey F. Naughton: "Query Execution Techniques for Caching Ex pensive Methods", Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canadá (junio, 1996).

25.19 Won Kim: "On Marrying Relations and Objects: Relation-Centric and Object-Centric Pers pectives", *Data Base Newsletter 22,* No. 6 (noviembre-diciembre, 1994).

Este artículo argumenta *en contra* de la posición de que la equiparación de varrels y clases es un gran error ("el primer gran error garrafal"). La referencia [25.12] es una respuesta a este artículo.

25.20 Won Kim: "Bringing Object/Relational Down to Earth", *DBP&D 10,* No. 7 (julio, 1997).

En este artículo, Kim dice que "seguramente reinará la confusión" en el mercado de objetos/relacional debido, en primer lugar, "a que se ha puesto un peso desproporcionado en el papel de la extensibilidad de los tipos de datos" y en segundo, a que "la medición de la entereza de objetos/relacional de un producto... es un área de perplejidad potencialmente seria". Continúa proponiendo "una medición práctica de la entereza de objetos/relacional que puede ser usada como un lineamiento para determinar si un producto en realidad es [de objetos/relacional]". Su esquema involucra los siguientes criterios:

1. Modelo de datos.
2. Lenguaje de consulta.
3. Servicios de misión crítica.
4. Modelo computacional.
5. Rendimiento y escalabilidad.
6. Herramientas de base de datos.
7. Aprovechamiento de la potencia.

Con respecto al primero de estos criterios (¡el crucial!) Kim toma la posición —muy diferente con respecto a la del *Tercer Manifiesto* [3.3]— de que el modelo de datos debe ser "el modelo de objetos central definido por el Grupo de Administración de Objetos" (OMG), el cual "comprende el modelo de datos relacional así como los conceptos centrales del modelado orientado a objetos de los lenguajes de programación orientados a objetos". De acuerdo con Kim, esto incluye todos los siguientes conceptos: *clase* (Kim añade "o tipo"), *ejemplar, atributo, restricciones de integridad, IDs de objetos, encapsulación, herencia de clase (múltiple), herencia de TAD (múltiple), datos de referencia de tipo, atributos con valor de conjunto, atributos de clase, métodos de clase* y muchos más. Observe que las relaciones —que por supuesto, nosotros vemos como cruciales y fundamentales— nunca son mencionadas explícitamente; Kim dice que el modelo de objetos central de OMG incluye al modelo relacional completo, además de todo lo anterior de la lista, aunque de hecho no es así.

25.21 Bruce Lindsay, John McPherson y Hamid Pirahesh: "A Data Management Extension Ar chitecture", Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif, (mayo, 1987).

Describe la arquitectura general del prototipo Starburst. Éste "facilita la implementación de extensiones de administración de datos para los sistemas de bases de datos relaciónales". En este artículo se describen dos tipos de extensiones: estructuras de almacenamiento y métodos de acceso definidos por el usuario, así como restricciones de integridad y procedimientos disparados definidos por el usuario (¿pero que no *todas* las restricciones de integridad son definidas por el usuario?). Sin embargo (para citar el artículo), "Existen, por supuesto, otras direcciones en las cuales es importante tener la capacidad de extender... los tipos de datos [y] las técnicas de evaluación de consultas definidas por el usuario [incluyendo al DBMS]".

25.22 Guy M. Lohman *et al.:* "Extensions to Starburst: Objects, Types, Functions, and Rules", *CACM34,* No. 10 (octubre, 1991).

25.23 David Maier: "Comments on the Third-Generation Database System Manifesto", Tech. Report No. CS/E 91-012, Oregon Graduate Center, Beaverton, Ore. (abril, 1991).

Maier es extremadamente crítico sobre casi todo lo que está en la referencia [25.34]. Estamos de acuerdo con algunas de sus críticas y en desacuerdo con otras. Sin embargo, encontramos interesantes los siguientes comentarios (ellos confirman nuestra pretensión de que los objetos involucran solamente una buena idea; es decir, el *soporte apropiado para los tipos de datos):* "Muchos de las personas que nos encontrarmos en el campo de las bases de datos orientadas a objetos, hemos luchado para destilar la esencia de la 'orientación a objetos' para un sistema de bases de datos... Mi propio pensamiento acerca de lo que era la característica más importante de las OODBs, ha cambiado a lo largo del tiempo. Al principio pensé que era la herencia y el modelo de mensajes. Después llegué a pensar que eran más importantes

la identidad de objetos, el soporte para el estado complejo y la encapsulación del comportamiento. Recientemente, después de comenzar a oír hablar a los usuarios de los OODBMSs sobre lo que para ellos es lo más valioso de esos sistemas, creo que la clave es la *extensibilidad de tipos.* La identidad, el estado complejo y la encapsulación siguen siendo importantes, pero [sólo] en tanto soporten la creación de nuevos tipos de datos."

25.24 Jignesh Patel *et al:* "Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

Para[1] citar el resumen: "Este artículo presenta varias técnicas nuevas para la paralelización de sistemas de bases de datos geoespaciales y trata su implementación en el sistema de base de datos de objetos/relacional Paradise" [25.15].

25.25 Raghu Ramakrishnan: *Database Management Systems.* Boston, Mass.: McGraw-Hill (1998).

25.26 Lawrence A. Rowe y Michael R. Stonebraker: "The Postgres Data Model", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

25.27 Hanan Samet: *The Design and Analysis of Spatial Data Structures.* Reading, Mass.: Addison-Wesley (1990).

25.28 Cynthia Maro Saracco: *Universal Database Management: A Guide to Object/Relational Tech nology.* San Francisco, Calif.: Morgan Kaufmann (1999).

Es una panorámica legible de alto nivel. Sin embargo, observamos que Saracco adopta (como lo hace Stonebraker en la referencia [25.31]) una forma muy sospechosa de herencia, involucrando una versión de la idea de subtablas y supertablas —sobre la cual en principio somos escépticos [13.12]— que es significativamente diferente con respecto a la versión incluida en SQL3. Para ser más específicos, suponga que la tabla PGMR ("programadores") está definida para que sea una subtabla de la tabla EMP ("empleados"). Entonces Saracco y Stonebraker ven a EMP como si contuviera filas sólo para los empleados que no son programadores y en cambio, SQL3 la vería como si contuviera filas para *todos* los empleados (vea el apéndice **B).**

25.29 Praveen Seshadri y Mark Paskin: "PREDATOR: An OR-DBMS with Enhanced Data Types", Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz, (mayo, 1997).

"La idea básica de PREDATOR es proporcionar mecanismos para cada tipo de datos con el fin de especificar la semántica de sus métodos; luego usa esta semántica para la optimización de consultas".

25.30 Michael Stonebraker: "The Design of the Postgres Storage System", Proc. 13th Int. Conf. on Very Large Data Bases, Brighton, Reino Unido (septiembre, 1987).

25.31 Michael Stonebraker and Paul Brown (with Dorothy Moore): *Object/Relational DBMSs: Tracking the Next Great Wave* (2nd edition). San Francisco, Calif.: Morgan Kaufmann (1999).

Este libro es un tutorial sobre sistemas de objetos/relaciónales. Está fuertemente basado —en realidad, casi exclusivamente— en la Opción de Datos Universal para el producto Dynamic Server de Informix. Esa opción de datos universal está basada en un sistema anterior llamado Illustra (un producto comercial donde el propio Stonebraker participó en su desarrollo). Vea la referencia [3.3] para un análisis y una crítica amplios sobre este libro; también vea el comentario a la referencia [25.28].

25.32 Michael Stonebraker y Greg Kemnitz: "The Postgres Next Generation Database Management System", *CACM34,* No. 10 (octubre, 1991).

25.33 Michael Stonebraker y Lawrence A. Rowe: "The Design of Postgres", Proc. 1986 ACM SIG MOD Int. Conf. on Management of Data, Washington, DC (junio, 1986).

Los objetivos declarados de Postgres son:

1. Proporcionar mejor soporte para objetos complejos.
2. Proporcionar extensibilidad al usuario para tipos de datos, operadores y métodos de acceso.
3. Proporcionar propiedades de bases de datos activas (alertadores y disparadores), así como soporte de inferencia.
4. Simplificar el código del DBMS para la recuperación de fallas.
5. Producir un diseño que pueda aprovechar los discos ópticos, las estaciones de trabajo con va rios procesadores y los chips VLSI diseñados en forma personalizada.
6 Hacer tan pocos cambios como sea posible (preferiblemente ninguno) al modelo relacional.

25.34 Michael Stonebraker *et al.:* "Third-Generation Database System Manifesto", *ACM SIGMOD Record 19,* No. 3 (septiembre, 1990).

En parte, este artículo es una respuesta —es decir, una contrapropuesta— al Manifiesto de los Sistemas de Bases de Datos Orientadas a Objetos [24.1], el cual (entre otras cosas) en esencia ignora por completo al modelo relacional (¡!). Una cita: "Los sistemas de segunda generación hicieron una gran contribución en dos áreas, el acceso a datos no procedural y la independencia de datos, y estos avances no deben ser comprometidos por los sistemas de tercera generación." Decimos que las siguientes características son requerimientos esenciales de un DBMS de tercera generación (hemos cambiado un poco las palabras originales):

1. Proporcionar los servicios de bases de datos tradicionales y además, estructuras de objetos y reglas más ricas
   - Sistema de tipos rico.
   - Herencia.
   - Funciones y encapsulación.
   - IDs de tupia asignados por el sistema, opcional.
   - Reglas (por ejemplo, reglas de integridad) que no estén atadas a objetos específicos.
2. Incluir los DBMSs de segunda generación
   - La navegación sólo como último recurso.
   - Definiciones de conjuntos intencionales y extensionales (lo que significa, colecciones que son mantenidas automáticamente por el sistema y colecciones que son mantenidas ma nualmente por el usuario).
   - Vistas actualizables.
   - Agolpamientos, índices, etcétera, ocultos ante el usuario.
3. Soporte de sistemas abiertos
   - Soporte de lenguajes múltiples.
   - Persistencia a los tipos.
   - SQL (caracterizado como "lenguaje de datos intergaláctico").
   - Las consultas y los resultados deben ser el nivel más bajo de la comunicación cliente-servi dor.

Vea la referencia [3.3] para un análisis y una crítica amplios de este artículo, así como también la referencia [25.23]. *Nota:* A propósito, ahora podemos explicar por qué *El Tercer Manifiesto* es llamado "tercer"... Fue escrito específicamente para que fuera la continuación (y esperamos superación) de los dos manifiestos anteriores, las referencias [24.1] y [25.34]

25.35 Maurice V. Wilkes: "Software and the Programmer", *CACM 34,* No. 5 (mayo, 1991).

APÉNDICES

El apéndice A proporciona más detalles (para propósitos de referencia) sobre la sintaxis y la semántica de las expresiones de SQL/92. El apéndice B da una panorámica de las principales características —especialmente las características de "objetos" o "de objetos/relaciónales"— de SQL3. El apéndice C presenta una lista de las abreviaturas y acronimos más importantes que presentamos en el cuerpo del texto, junto con su significado.

APÉNDICE A

Expresiones SQL

A.l INTRODUCCIÓN

Las expresiones SQL —para ser más específicos, las expresiones **tabla, condicionales y escalares** de SQL— son el corazón del lenguaje SQL. En este apéndice presentamos una descripción detallada de la sintaxis y semántica de dichas expresiones (tal como son en SQL/92). Sin embargo, debemos decir inmediatamente que los nombres que usamos para las categorías sintácticas y las construcciones del lenguaje SQL son, en su mayor parte, diferentes de las que se usan en el propio estándar [4.22], debido a que los términos del estándar a menudo no son muy adecuados; en realidad, los mismos términos *expresión de tabla, expresión condicional* y *expresión escalar* no son términos estándar.

A.2 EXPRESIONES DE TABLA

Aquí primero tenemos una gramática BNF para las *<expresiones de tabla>.* La gramática está completa, con excepción de unas cuantas opciones que tienen que ver con los nulos (vea el capítulo 18, sección 18.7). Observe que hacemos un gran uso de la convención de lista separada con comas que presentamos en el capítulo 4, sección 4.6.

```
<expresión de tabla>
   ::=  <expresión de tabla con junta>
      | <expresión de tabla sin junta>

<expresión de tabla con junta>
   ::=  <referencia de tabla> [ NATURAL ] JOIN <referencia de tabla>
                    [ ON <expresión condicional>
                    | USING ( <lista de nombres de columna separados con comas> ) ]
        <referencia de tabla> CROSS JOIN <referencia de tabla> ( <expresión de tabla
        con junta> )

<referencia de tabla>
   ::=  <nombre de tabla> [ [ AS ] <nombre de variable de alcance>
                          [ ( <lista de nombres de columna separados con comas> ) ] ] |
      ( <expresión de tabla> ) [ AS ] <nombre de variable de alcance>
                               [ ( <lista de nombres de columna separados con comas> ) ]
      | <expresión de tabla con junta>

<expresión de tabla sin junta>
   ::=  <término de tabla sin junta>
      | <expresión de tabla> UNION [ ALL ]
        [ CORRESPONDING [ BY ( <lista de nombres de columnas separados con comas> ) ] ]
                    <término de tabla>
```

```
       I <expresión de tabla> EXCEPT [ ALL ]
         [ CORRESPONDING [ BY ( <lista de nombres de columna separados con comas> ) ] ]
                    <término de tabla>

<término de tabla sin junta>
   ::=  <primaria de tabla sin junta>
       | <término de tabla> INTERSECT [ ALL ]
         [ CORRESPONDING [ BY ( <lista de nombre de columna separados con comas> ) ] ]
                    <primaria de tabla>

<término de tabla>
   ::=  <término de tabla sin junta> |
       <expresión de tabla con junta>

<primaria de tabla>
   ::=  <primaria de tabla sin junta> |
       <expresión de tabla con junta>

<primaria de tabla sin junta> ::=  TABLE
   <nombre de tabla> <constructor de tabla>
   <expresión de seleccionara ( <expresión de
   tabla sin junta> )

<constructor de tabla>
   ::=  VALUES <lista de constructores de fila separados con comas>

constructor de fila>
         <expresión escalar*
         ( <lista de expresiones escalares separadas con comas> )
         ( <expresión de tabla> )

<expresión de seleccionar>
   ::=  SELECT [ ALL | DISTINCT ] <lista de elementos de seleccionar separados con comas>
           FROM <lista de referencias de tabla separadas con comas> [ WHERE <expresión
           condicional> ]
                 [ GROUP BY <lista de nombres de columna separados con comas> ]
                    [ HAVING <expresión condicional> ]

<elemento de seleccionar>
   :: =  <expresión escalar> [ [ AS ] <nombre de columna> ]
       | [ <nombre de variable de alcance>. ] *
```

Ahora desarrollaremos un caso especial (probablemente el más importante en la práctica): es decir, <*expresión de seleccionar*>. Una <*expresión de seleccionaré* puede ser vista, aproximadamente, como una <*expresión de tabla*> que no involucra JOINs, UNIONs, EXCEPTs o INTERSECTS —"aproximadamente" debido a que, por supuesto, tales operadores podrían estar involucrados en expresiones que estén anidadas dentro de la <*expresión de seleccionaré* en cuestión. Para una exposición de JOIN, UNION, EXCEPT e INTERSECT vea el capítulo 7, sección 7.7.

Como lo indica la gramática anterior, una <*expresión de seleccionaré*- involucra, de manera secuencial, una cláusula SELECT, una cláusula FROM y (opcionalmente) una cláusula WHERE, una cláusula GROUP BY y una cláusula HAVING. Consideraremos a las cláusulas una por una.

La cláusula SELECT

La cláusula SELECT toma la forma

```
SELECT [ ALL | DISTINCT ] <lista de elementos de seleccionar separados con comas>
```

Explicación:

1. La <*lista de elementos de seleccionar separados con comasr*> no debe estar vacía (vea más adelante para una exposición detallada de <*elemento de seleccionar>).*
2. Si no se especifica ALL ni DISTINCT, se toma a ALL.
3. Suponga, por el momento, que ya han sido evaluadas las cláusulas FROM, WHERE, GROUP BY y HAVING. Sin importar cuáles de estas cláusulas se hayan especificado y cuáles se hayan omitido, el resultado conceptual de su evaluación siempre es una tabla (po siblemente una tabla "agrupada"; vea más adelante) a la cual nos referiremos como tabla 77 (aunque ese resultado conceptual en realidad no tiene nombre).
4. Sea 72 la tabla que es derivada, a partir de 77, por medio de la evaluación de los <*elementos de seleccionar*> especificados contra 77 (vea más adelante).
5. Sea *T3* la tabla que es derivada, a partir de 72, mediante la eliminación de filas duplicadas re dundantes de 72 si se especifica DISTINCT, o una tabla que es idéntica a 72 en caso contrario.
6. *T3* es el resultado final.

Pasemos ahora a una explicación de los <*elementos de seleccionara.* Hay dos casos a considerar, de los cuales el segundo es simplemente una abreviatura para una lista separada con comas de <*elemento de seleccionar*>, de la primera forma; por lo tanto, el primer caso es en realidad el más fundamental.

Caso 1: El <*elemento de seleccionaré* toma la forma

```
<expresión escalar>  [  [ AS ]  <nombre de columna> ]
```

La <*ekpresión escalar*> involucrará, por lo general (aunque no necesariamente), una o más columnas de 77 (vea el párrafo 3 anterior). Para cada fila de *TI,* la <*expresión escalar*> es evaluada para producir un resultado escalar. La lista separada con comas de tales resultados (que corresponden con lá evaluación de todos los <*elementos de seleccionar*> de la cláusula SELECT contra una sola fila de 77) cpnstituye una sola fila de 72 (vea el párrafo 4 anterior). Si el <*elemento de seleccionar*> incluye una cláusula AS> el <*nombre de columna*> no calificado de esa cláusula es asignado como nombre de la columna correspondiente de 72 (la palabra opcional reservada AS es simplemente ruido, y puede ser omitida sin afectar el significado). Si el <*elemento de seleccionar*> no incluye una cláusula AS, entonces (a) sí consiste simplemente en un <*nombre de columna*> (posiblemente calificado), entonces ese <*nombre de colwnna*> es asignado como nombre de la columna correspondiente de 72; (b) en caso contrario la columna correspondiente de 72 en efecto no tiene nombre (en realidad se le da un nombre "dependiente de la implementación" [4.19], [4.22]). Surgen estos puntos:

- Debido a que se trata específicamente del nombre de una columna de 72, y no de *TI,* un nombre introducido por medio de una cláusula AS no puede ser usado en las cláusulas WHERE, GROUP BY ni HAVING (en caso de haberlas) involucradas directamente en la construcción de *TI.* Sin embargo, sí puede ser referida en una cláusula ORDER BY asociada (en caso de haberla) —en particular en DECLARE CURSOR— y también en una <*expresión de tabla*> "externa" que contenga la <*expresión de seleccionar*> que está siendo explicada anidada dentro de ella.
- Si un <*elemento de seleccionaré* incluye una llamada a un operador de totales *y* la <*expresión de seleccionara* no incluye una cláusula GROUP BY (vea más adelante), entonces ningún <*elemento de seleccionara* en la cláusula SELECT puede incluir una referencia a

una columna de *TI,* a menos que esa referencia de columna sea el argumento (o parte del argumento) de una llamada a un operador de totales. *Caso 2:* El <*elemento de seleccionara* toma la forma

[<nombre de variable de alcance> .] *

Si el calificador es omitido (es decir, el <*elemento de selecciona*» es simplemente un asterisco no calificado), entonces este <*elemento de selecciona*» debe ser el único <*elemento de selecciona*» en la cláusula SELECT. Esta forma es una abreviatura para una lista separada con comas de todos los <*nombres de columna*> de 77, en un orden de columnas de izquierda a derecha. Si el calificativo está incluido (es decir, el <*elemento de selecciona*» consiste en un asterisco calificado por una variable de alcance llamada *R;* por lo tanto: "/?.*"), entonces el <*elemento de selecciona*» representa una lista separada con comas de <*nombres de columna*> para todas las columnas de la tabla asociada con la variable de alcance *R,* en un orden de izquierda a derecha. (Recuerde de la sección 7.7 que un nombre de tabla puede ser usado (y a menudo así será) como una variable de alcance implícita. Por lo tanto, el <*elemento de selecciona*» con frecuencia será de la forma 'T.*" en lugar de "/?.*").

La cláusula FROM

La cláusula FROM toma la forma

FROM <lista de referencias de tabla separadas con comas>

La <*lista separada con comas de referencias de tabla* > no debe estar vacía. Sea que las <*referencias de tabla*> den como resultado las tablas *A, B,..., C,* respectivamente. Entonces el resultado de la evaluación de la cláusula FROM es una tabla que es igual al producto cartesiano de *A, B,..., C. Nota:* Recuerde que el producto cartesiano de una sola tabla Testa definido para que sea igual a *T* (vea la respuesta al ejercicio 6.12 del capítulo 6); en otras palabras, es válido (por supuesto) que la cláusula FROM contenga una sola <*referencia de tabla*>.

La cláusula WHERE

La cláusula WHERE toma la forma

WHERE <expresión condicional>

Sea *T* el resultado de la evaluación de la cláusula FROM inmediata anterior. Entonces el resultado de la cláusula WHERE es una tabla que está derivada de *T* mediante la eliminación de todas las filas para las cuales la <*expresión condicional*> no da como resultado *verdadero.* Si la cláusula WHERE es omitida, el resultado es simplemente *T.*

La cláusula GROUP BY

La cláusula GROUP BY toma la forma

GROUP BY <lista de nombres de columna separados con comas>

La <*lista de nombre de columna separados con comas*> no debe estar vacía. Sea *T* el resultado de la evaluación de las cláusulas FROM y WHERE (en caso de haberla) inmediatas anteriores.

Cada <*nombre de columna*> mencionado en la cláusula GROUP BY debe ser el nombre, opcionalmente calificado, de una columna de *T*. El resultado de la cláusula GROUP BY es una **tabla agrupada;** es decir, un conjunto de grupos de filas derivados de *T*mediante el reacomodo conceptual de las filas en la mínima cantidad de grupos tales que, dentro de cualquier grupo, todas las filas tengan el mismo valor para la combinación de columnas identificadas en la cláusula GROUP BY. Por lo tanto, observe cuidadosamente que el resultado es entonces "una tabla no auténtica" (para repetir, es una tabla de grupos, no una tabla de filas). Sin embargo, una cláusula GROUP BY nunca aparece sin una cláusula SELECT correspondiente, cuyo efecto es derivar una tabla auténtica (es decir, una tabla de filas) a partir de esa tabla de grupos, y por lo tanto, no hace mucho daño al ser apartada temporalmente del marco relacional.

Si una <*expresión de seleccionara* incluye una cláusula GROUP BY, entonces hay restricciones sobre la forma que puede tomar la cláusula SELECT correspondiente. Para ser específicos, cada <*elemento de seleccionaré* en la cláusula SELECT (incluyendo cualquiera que esté implicado por la abreviatura de asterisco) debe ser uno con **un solo valor por grupo.** Por lo tanto, tal <*elemento de seleccionaré* no debe incluir ninguna referencia a cualquier columna de la tabla *T* que no esté mencionada en la propia cláusula GROUP BY, *a menos* que esa referencia sea el argumento —o parte del argumento— de alguna llamada a un operador de totales (por supuesto, el efecto de dicha llamada es reducir alguna colección de valores escalares de un grupo a un solo valor escalar).

La cláusula HAVING

La cláusula HAVING toma la forma

```
HAVING <expresión condicional>
```

Sea *G* la tabla agrupada resultante de la evaluación de las cláusulas FROM, WHERE (en caso de haberla) y GROUP BY (en caso de haberla) inmediatas anteriores. Si no hay una cláusula GROUP BY, entonces tomamos a *G* como el resultado de la evaluación de las cláusulas FROM y WHERE solas, y es considerada como una tabla agrupada que contiene sólo un grupo;* en otras palabras, en este caso hay una cláusula GROUP BY conceptual implícita que especifica que *no hay en absoluto columnas de agrupamiento.* El resultado de la cláusula HAVING es una tabla agrupada que es derivada de *G* mediante la eliminación de todos los grupos para los cuales la <*expresión condicional*> no da como resultado *verdadero.* Surgen estos puntos:

- Si la cláusula HAVING es omitida, pero se incluye la cláusula GROUP BY, el resultado de la cláusula HAVING es simplemente *G.* Si las cláusulas HAVING Y GROUP BY son omi tidas, el resultado es simplemente la tabla *T* "auténtica" —es decir, no agrupada— resul tante de las cláusulas FROM y WHERE.
- Cualquier <*expresión escalar*> que esté en una cláusula HAVING, debe producir un valor único por grupo (al igual que la <*expresión escalar*> de la cláusula SELECT, si hay una cláusula GROUP BY, como dijimos anteriormente).

---

*Esto es lo que dice el estándar, aunque lógicamente debería decir un grupo *como máximo* (si las cláusulas FROM y WHERE producen una tabla vacía, no debe haber ningún grupo).

Un ejemplo completo

Concluimos nuestra explicación de <*expresión de seleccionaré* con un ejemplo razonablemente complejo que ilustra algunos de (pero no todos) los puntos que explicamos anteriormente. La consulta es la siguiente: *para todas las partes rojas y azules tales que la cantidad total suministrada sea mayor a 350 (excluyendo del total todos los envíos en los que la cantidad sea menor o igual a 200), obtener el número de parte, el peso en gramos, el color y la cantidad máxima suministrada de esa parte.* Aquí tenemos una formulación SQL posible de esta consulta:

```
SELECT P.P#,
       'Peso en gramos =' AS TEXT01,
       P.PESO * 454 AS PSGR,
       P.COLOR,
       'Cantidad máxima = ' AS TEXT02,
       MAX ( VP.CANT ) AS CANTMAX
FROM  P, VP WHERE  P.P# = VP.P#
AND  ( P.COLOR = 'Rojo' OR P.COLOR ■ 'Azul' )
AND   VP.CANT > 200 GROUP BY P.P#, P.PESO,
P.COLOR HAVING SUM ( VP.CANT ) > 350 ;
```

Explicación: Primero observe que (como explicamos en las subsecciones anteriores) las cláusulas de una <*expresión de seleccionaré-* son evaluadas conceptualmente en el orden en que están escritas; con la única excepción de la propia cláusula SELECT, la cual es evaluada hasta el final. Por lo tanto, en el ejemplo podemos imaginar el resultado que está siendo construido de la siguiente forma:

1. **FROM.** La cláusula FROM es evaluada para producir una nueva tabla que es el producto cartesiano de las tablas **P** y VP.
2. **WHERE.** El resultado del paso 1 es reducido mediante la eliminación de todas las filas que no satisfacen la cláusula WHERE. Por lo tanto, en el ejemplo son las filas que no satisfacen la <*expresión condicional*>

   ```
          p.p# = VP.P#
   AND  ( P.COLOR = 'Rojo'  OR P.COLOR » 'Azul' )
   AND   VP.CANT > 200
   ```

3. **GROUP BY.** El resultado del paso 2 es agrupado de acuerdo con los valores de las colum nas mencionadas en la cláusula GROUP BY. En el ejemplo, esas columnas son P.P#, P.PESO y P.COLOR. *Nota:* En teoría, P.P# por sí solo podría ser suficiente como columna de agrupamiento, debido a que P.PESO y P.COLOR son, en sí mismos, de un solo valor por número de parte (es decir, son funcionalmente dependientes del número de parte). Sin embargo, SQL no está consciente de este último hecho y producirá un error si P.PESO y P.COLOR son omitidos en la cláusula GROUP BY, debido a que *son* mencionados en la cláusula SELECT. Vea la referencia [10.6] para una explicación de este punto.
4. **HAVING.** Los grupos que no satisfacen la <*expresión condicionaba*

   ```
   SUM ( VP.CANT ) > 350
   ```

   son eliminados del resultado del paso 3.
5. **SELECT.** Cada grupo en el resultado del paso 4 genera una sola fila de resultado, de la siguiente manera. Primero, el número de parte, el peso, el color y la cantidad máxima, son

extraídos del grupo. Segundo, el peso es convertido a gramos. Tercero, las dos cadenas de caracteres "Peso en gramos =" y "Cantidad máxima =", son insertadas en los lugares adecuados de la fila. A propósito, observe que —como lo sugiere la frase "los lugares adecuados de la fila"— nos estamos apoyando en el hecho de que las columnas de las tablas tienen en SQL un ordenamiento de izquierda a derecha; las cadenas no tendrían mucho sentido si no aparecieran en esos "lugares adecuados".

El resultado final se ve de esta forma:

p#	TEXTO1	PESOGRAMOS	COLOR	TEXT02	CANTMAX
P1	Peso en gramos =	5448	Rojo	Cantidad máxima ■	308
P5	Peso en gramos ■	5448	Azul	Cantidad máxima ■	400
P3	Peso en gramos =	7718	Azul	Cantidad máxima =	400

Comprenda que el algoritmo que acabamos de escribir pretende ser simplemente una explicación **conceptual** de la manera en que es evaluada una <*expresión de seleccionara.* El algoritmo en realidad es correcto en el sentido de que garantiza que se produzca el resultado correcto. Sin embargo, probablemente sería bastante ineficiente si en realidad fuera ejecutado de esta manera. Por ejemplo, sería muy desafortunado si el sistema en realidad construyera el producto cartesiano en el paso 1. Consideraciones como éstas son precisamente la razón por la cual los sistemas relaciónales requieren de un optimizador, como mencionamos en el capítulo 17. Además, la tarea del optimizador en un sistema SQL puede ser caracterizada como la localización de un procedimiento de implementación que producirá el mismo resultado que el algoritmo conceptual que esbozamos anteriormente, pero que es más eficiente que ese algoritmo.

A.3 EXPRESIONES CONDICIONALES

Al igual que <*expresión de tabla*>, el término <*expresión condiciónate* aparece en numerosos contextos por todo el lenguaje SQL; en particular se usa por supuesto en las cláusulas WHERE para calificar o descalificar filas para su procesamiento posterior.* En esta sección tratamos algunas de las características más importantes de dichas expresiones condicionales. Sin embargo, observe que nuestro tratamiento *no* pretende en definitiva ser completo. En particular, ignoramos nuevamente todo lo referente a los nulos; como vimos en el capítulo 18, la <*expresión condiciónate* requiere de un tratamiento muy amplio cuando se toman en cuenta las implicaciones y complicaciones de los nulos, y hemos proporcionado determinados formatos de <*expresión condiciónate,* que no tratamos en este apéndice, con el único fin de manejar ciertos aspectos del soporte a nulos. Sin embargo, estos aspectos fueron descritos en el capítulo 18.

Al igual que la sección anterior, comenzamos con una gramática BNF. Luego pasamos a explicar con mayor detalle determinados casos específicos; es decir <*condición de semejanza*>, <*condición de coincidencia*>, <*condición de todos o cualquiera*> y <*condición de unicidad*> (todos los demás casos ya los hemos ilustrado en el cuerpo del libro o son muy fáciles de entender).

---

*Le recordamos que (como indicamos en el capítulo 8) las expresiones condicionales son el equivalente de SQL para lo que hemos estado llamando expresiones *lógicas* en el cuerpo del libro.

```
<expresión condicional>
   ::=  <término condicional>
      | <expresión condicional> OR <término condicional>

<término condicional>
   ::=  <factor condicional>
      | <término condicional> AND <factor condicional>

<factor condicional>
   ::■  [ NOT ] <primario condicional>

<primario condicional>
   ::=  <condición simple> | ( <expresión condicionad )

<condición simple>
   ::=  <condición de comparación>
        <condición de pertenencia>
        <condición de semejanza> <condición
        de coincidencia> <condición de todos
        o cualquiera> <condición de
        existencia> <condición de unicidad>

<condición de comparación> ::=
   constructor de fila>
               <operador de comparación> <constructor de fila>

<operador de comparación>
   ::=  = | < | <= | > | >= | <>

<condición de pertenencia>
   ::=  <constructor de fila> [ NOT ] IN ( <expresión de tabla> ) |
      <expresión escalar> [ NOT ] IN
                          ( <lista de expresiones escalares separadas con comas> )

<condición de semejanza>
   ::=  <expresión de cadena de caracteres> [ NOT ] LIKE <patrón>
                                            [ ESCAPE <escape> ]

<condición de coincidencia>
   ::=  <constructor de fila> MATCH UNIQUE ( <expresión de tabla> )

<condición de todos o cualquiera>
   ::=  constructor de fila>
           <operador de comparación> ALL ( <expresión de tabla> )
      | <constructor de fila>
           <operador de comparación> ANY ( <expresión de tabla> )

condición de existencia>
   ::=  EXISTS ( «expresión de tabla> )

condición de unicidad>
   ::=  UNIQUE ( <expresión de tabla> )
```

Condiciones LIKE

Las condiciones LIKE están orientadas para una concordancia simple de patrones sobre cadenas de caracteres; es decir, prueban una cadena de caracteres dada para ver si se apega a algún patrón prescrito. La sintaxis (para repetir) es :

```
<expresión de cadena de caracteres> [ NOT ] LIKE <patrón>
                                              [ ESCAPE <escape> ]
```

Aquí <*patrón*> es una expresión de cadena de caracteres cualquiera y <*escape*> (en caso de que sea especificada) es una expresión de cadena de caracteres que da como resultado un solo carácter. Éste es un ejemplo:

```
SELECT P.P#, P.PARTE
FROM   P
WHERE  P.PARTE LIKE 'LV ;
```

("Obtener los números y nombres de parte para las partes cuyos nombres comienzan con la letra L"). *Resultado:*

p#	PARTE
P5 P6	Leva cog

Mientras no sea especificada una cláusula ESCAPE, los caracteres que están dentro de <*patrón*> se interpretan de la siguiente manera:

- El carácter de subrayado "_" significa *cualquier carácter solo.*
- El carácter de por ciento "%" significa *cualquier secuencia de n caracteres* (en donde *n* puede ser cero).
- Todos los demás caracteres significan los que son ellos mismos.

Por lo tanto, en el ejemplo, la consulta regresa filas de la tabla P en las cuales el valor de PARTE comienza con una letra L mayúscula y tiene cualquier secuencia de cero o más caracteres a continuación de esa L.

Éstos son otros ejemplos:

```
DOMICILIO LIKE '%Berkeley%'
```
da como resultado *verdadero* si DOMICILIO contiene la cadena "Berkeley" en cualquier lugar dentro de ella

```
V# LIKE 'V
```
da como resultado *verdadero* si V# es de exactamente 3 caracteres de largo y el primero es "V"

```
PARTE LIKE '%C
```
da como resultado *verdadero* si PARTE es de 4 caracteres de largo o más y el anterior al antepenúltimo es "C"

```
MITEXTO LIKE '-_%'
        ESCAPE "-'
```
da como resultado *verdadero* si MITEXTO comienza con un carácter de subrayado (vea más adelante)

En este último ejemplo, el carácter "=" ha sido especificado como carácter de escape, y esto significa que, si se desea, puede ser desactivado el significado especial que se da a los caracteres "_" y "%", antecediendo dichos caracteres con un carácter "=". Por último, la <*condición de semejanza*>

```
X NOT LIKE y [ ESCAPE z ]
```

está definida para que sea semánticamente equivalente a

```
NOT ( x LIKE y [ ESCAPE z ] )
```

Condiciones MATCH

Una <*condición de coincidencico* toma la forma

```
constructor de fila> MATCH UNIQUE ( <expresión de tabla> )
```

Sea *rl* la fila que resulta de la evaluación de <*constructor defila*> y sea Tía tabla que resulta de la evaluación de <*expresión de tabla*>. Entonces la <*condición de coincidencia*> da como resultado *verdadero* si y sólo si *T* contiene exactamente una fila, digamos *r2,* tal que la comparación

r1 = ***r2*** dé como resultado *verdadero.* Éste es

un ejemplo:

```
SELECT VP.«
FROM   VP
WHERE  NOT ( VP.V# MATCH UNIQUE ( SELECT V.V# FROM V ) ) ;
```

("Obtener los envíos que no tienen exactamente un proveedor coincidente en la tabla de proveedores".) Dicha consulta podría ser útil en la verificación de la integridad de la base de datos, debido a que, por supuesto, no *deberá* haber ninguno de estos envíos si la base de datos es correcta. Sin embargo, observe que podríamos usar una <*condición de pertenencia*> para realizar exactamente la misma verificación.

A propósito, UNIQUE puede ser omitida de MATCH UNIQUE, pero entonces MATCH se convierte en sinónimo de IN (al menos en ausencia de nulos).

Condiciones de todos o cualquiera

Una <*condición de todos o cualquiera*> tiene la forma general

```
constructor de fila> <operador de comparación> <calificador>
                                                ( <expresión de tabla> )
```

donde el <*operador de comparación*> es cualquier elemento del conjunto usual (=, o, etcétera) y el <*calificador*> es ALL o ANY. En general, una *Kcondición de todos o cualquiera*> da como resultado *verdadero* si y sólo si la comparación correspondiente sin el ALL (ANY, respectivamente) da como resultado *verdadero* para todas las filas (para cualquiera, respectivamente) de la tabla representada por la <*expresión de tabla*>. (Si esa tabla está vacía, las condiciones ALL dan como resultado *verdadero* y las condiciones ANY dan como resultado *falso.)* Éste es un ejemplo: "obtener los nombres de parte para las partes cuyo peso sea mayor que el de cada parte azul"

```
SELECT DISTINCT PX.PARTE
FROM  P AS PX
WHERE  PX.PESO >ALL ( SELECT PY.PESO
                      FROM  P AS PY
                      WHERE PY.COLOR - 'Azul' ) ;
```

Dados nuestros datos de muestra usuales, el resultado se ve como sigue:

EPARTE
Engrane

Explicación: La <*expresión de tabla*> anidada regresa el conjunto de pesos de las partes azules. Después, el SELECT externo regresa el nombre de esas partes cuyo peso es mayor que cada valor que está en ese conjunto. En general, por supuesto, el resultado final puede contener cualquier cantidad de nombres de parte (incluyendo cero).

Nota: Aquí vale la pena un comentario de precaución, al menos para quienes tienen al inglés como lengua materna. El hecho es que la <*condición de todos o cualquiem*> es muy propensa a error. Una formulación en inglés muy natural de la consulta anterior usaría la palabra "any" en vez de "every", lo cual podría fácilmente conducir al uso incorrecto de >ANY en vez de >ALL. Una crítica similar se aplica a todos los usos de los operadores ANY y ALL.

Condiciones UNIQUE

Una <*condición de unicidad*> se usa para probar que cada fila dentro de alguna tabla sea única (es decir, que no haya duplicados). La sintaxis es:

```
UNIQUE ( <expresión de tabla> )
```

La condición da como resultado *verdadero* si la <*expresión de tabla*> da como resultado una tabla en la cual todas las filas son distintas, *y falso* en caso contrario.

A.4 EXPRESIONES ESCALARES

Las <*expresiónes escalares*> de SQL son esencialmente directas. Aquí nos contentamos con una lista de algunos de los operadores más importantes que pueden ser usados en la construcción de dichas expresiones y proporcionamos simplemente algunos comentarios sobre un par de estos operadores (CASE y CAST) cuyo significado tal vez no sea inmediatamente aparente. En orden alfabético los operadores son:

```
operadores aritméticos (+, -, *, /)
BITJ-ENGTH
CASE
CAST
CHARACTERJ.ENGTH
concatenación (||)
CURRENTJJSER
LOWER
OCTETLENGTH
POSITION
SESSIONUSER
SUBSTRING
SYSTEMJJSER
TRIM
UPPER
USER
```

Observe que las llamadas de operadores de totales también pueden aparecer dentro de una <*expresión escalar*>, debido a que regresan un resultado escalar. Además, una <*expresión de tabla*> encerrada entre paréntesis también puede ser tratada como un valor escalar, siempre y cuando dé como resultado una tabla de exactamente una fila y una columna.

Ahora profundizaremos un poco más en los operadores CASE y CAST.

Operaciones CASE

Una operación CASE regresa un valor de un conjunto de valores especificado, dependiendo de la condición especificada. Por ejemplo:

```
CASE
  WHEN  V .STATU  <  5  THEN  'Última
  WHEN  V .STATU  < 10  THEN  'Dudoso'
  WHEN  V .STATU  < 15  THEN  'No muy bueno'
  WHEN  V .STATU  < 20  THEN  'Mediocre'
  WHEN  V .STATU  < 25  THEN  'Aceptable'
  ELSE                        'Bueno'
END
```

Operaciones CAST

CAST convierte un valor escalar especificado en un tipo de datos escalar especificado. Por ejemplo:

```
CAST ( P.PESO AS FLOAT )
```

No todos los pares de tipos de datos son convertibles mutuamente; por ejemplo, no son soportadas las conversiones entre números y cadenas de bits. Vea la referencia [4.22] para detalles de precisamente qué tipos de datos pueden ser convertidos en qué otros.

APÉNDICE **B**

Una panorámica de SQL3

B.l INTRODUCCIÓN

Para cuando este libro aparezca impreso, SQL3 será probablemente ratificado como un estándar ("SQL/99")- Sin embargo, no quisimos basar nuestras explicaciones anteriores sobre SQL en el SQL3; en parte por las razones que explicamos en el capítulo 4 y en parte debido a que —por no decir más— encontramos que SQL3 es algo confuso. Por otra parte, sentimos que en algún lado debía incluirse una panorámica, al menos de las características principales, del SQL3 y por eso es este apéndice.

Por supuesto, SQL3 incluye al todo SQL/92, con excepción de algunas características menores (que no tratamos en este libro) que han sido eliminadas deliberadamente: SQLCODE, enteros sin signo en lugar de nombres de columnas en ORDER BY, "introductores" sobre identificadores, determinados conjuntos de caracteres, secuencias de ordenamiento y traducciones definidas por el usuario, así como algunas otras cosas. Sin embargo, por razones obvias, aquí nos concentramos en las características que han sido agregadas desde 1992, y usamos por conveniencia el nombre "SQL3" para referirnos específicamente a esas características. De esas nuevas características, las más importantes son seguramente las que tienen que ver con los tipos de datos definidos por el usuario y otros aspectos relacionados, los cuales tratamos en las secciones B.2 a B.5. De las demás características, las más importantes son analizadas brevemente en la sección B6. *Nota:* En la referencia [3.3], puede encontrar un análisis y una crítica detallados del tema de las secciones B.2 a B.5 .

Antes de entrar en detalles sobre el propio SQL3, debemos decir unas palabras con relación a **SQLJ** [4.6]. El nombre "SQLJ" se refiere, de manera informal, a un proyecto para considerar posibles grados de integración entre SQL y Java (el proyecto es un esfuerzo conjunto que involucra a algunos de los fabricantes más conocidos de DBMS SQL). La parte 0 de ese proyecto trata sobre SQL incrustado en programas Java, la parte 1 trata la idea de llamar a Java desde SQL (por ejemplo, llamar un procedimiento almacenado que esté escrito en Java) y la parte 2 trata la posibilidad de usar las clases de Java como tipos de datos SQL (por ejemplo, como base para la definición de columnas en tablas SQL). Ninguna de estas actividades es técnicamente parte de SQL3 como tal, pero la parte 0 del SQLJ ya ha sido publicada (al menos en los EUA) como el primer componente de una nueva *parte 10* del borrador para el estándar de SQL [4.22], y es probable que pronto la sigan las partes 1 y 2 de SQLJ, posiblemente por el tiempo en que se publique formalmente SQL3.

También es probable que aparezca una nueva versión de **SQL/CLI** (vea el capítulo 4) para entonces.

Haremos algunos comentarios editoriales con relación a lo que viene a continuación: Primero, con frecuencia usamos el carácter "#" en los nombres de columna de nuestros ejemplos (como hicimos en el cuerpo del libro) aunque "#" no es de hecho un carácter válido en SQL3;

también usamos el punto y coma "j" para finalizar instrucciones (de nuevo, igual que en los ejemplos anteriores). Segundo, debemos dejar claro que las explicaciones que vienen a continuación de ninguna forma son completas. Finalmente, es natural que algunos detalles podrían haber cambiado para cuando se ratifique SQL3. Tenga presenta esto último.

B.2 NUEVOS TIPOS DE DATOS

Como sugerimos en la sección anterior, los aspectos más visibles inmediatamente de SQL3 tienen que ver con los tipos de datos.* Hay nuevos tipos **integrados** (nuevos tipos *escalares* integrados, para ser más precisos), hay nuevos **generadores de tipo** integrados (SQL3 los llama *constructores* de tipo) y hay una instrucción **CREATE TYPE** que permite que los usuarios definan sus propios tipos (y por supuesto, también una instrucción DROP TYPE correspondiente). Consideraremos, en su momento, a cada una de estas características.

Tipos escalares integrados

Están soportados tres nuevos tipos escalares integrados:

- **BOOLEAN.** Por supuesto, BOOLEAN es un tipo de *valor de verdad;* los operadores lógicos usuales (NOT, AND, OR) están soportados y una expresión lógica puede aparecer en todos los lugares donde pueden aparecer normalmente las expresiones escalares (en términos generales). Sin embargo, observe que SQL considera que hay *tres* valores de verdad y no dos, como vimos en el capítulo 18; las literales correspondientes son TRUE, FALSE y UNKNOWN. A pesar de este hecho, el tipo BOOLEAN incluye sólo dos valores y no tres; el valor de verdad *desconocido* es representado —¡de manera bastante incorrecta!— por *nulo* (por ejemplo, la asignación de UNKNOWN para una variable de tipo BOOLEAN en realidad pondr| esa variable igual a nulo). Para comprender la seriedad de esta falla, tal vez tenga que meditar en la equivalencia de un tipo numérico que usa nulo en lugar de cero para representar cero.

  Otra *rareza* es que SQL3 (extrañamente) no considera una referencia de variable lógica simple, como un ejemplar de lo que en el apéndice A llamamos <*primario condicional*>. Entonces, si B es por ejemplo de tipo BOOLEAN, ¡una cláusula WHERE que tenga la forma WHERE B no es válida!

  Junto con el tipo BOOLEAN, SQL3 presenta dos nuevos *operadores de totales:* EVERY —no ALL, por alguna razón— y ANY. El argumento en ambos casos es una columna de valores BOOLEAN (casi seguramente una columna derivada, como en —por ejemplo— la cláusula WHERE ANY (CANT > 200)). Si la columna está vacía, ambos operadores regresan *desconocido* (o más bien, *nulo);*[1]' si la columna no está vacía, EVERY regresa *verdadero* cuando todos los valores de la columna son *verdaderos (y falso* en caso contrario) y ANY regresa/a/so cuando todos los valores de la columna son *falsos* (y *verdadero* en caso contrario). Igual que como sucede con otros operadores de totales de SQL, los nulos son eliminados antes de realizar la incorporación.

---

*Es tentador agregar que una de las partes más «visibles inmediatamente son los "dominios" al estilo SQL (vea los capítulos 4 y 5), los cuales parecen haber sido ignorados calladamente.

[t]A este respecto, compare el comportamiento de las condiciones de todos o cualquiera (vea el apéndice A).

- **CLOB** ("objeto de caracteres grande"). Este tipo representa esencialmente cadenas de tamaño ilimitado con caracteres de longitud variable. Un mecanismo *localizador* asociado, similar (en cierta forma) al mecanismo de cursor familiar, permite que tales cadenas sean accedidas por partes. Muchos de los operadores de cadenas de caracteres usuales no están soportados para estas cadenas; entre los que sí *están* soportados se encuentran "=" y LIKE.
- **BLOB** ("objeto binario grande"). Son equivalentes, con excepción de que las cadenas en realidad son de "octetos" —es decir, bytes— en lugar de caracteres.

Tipos generados

Los generadores de tipo de SQL3 son REF, ARRAY y ROW. Sin embargo, la única forma para definir un "tipo REF' es en forma *implícita,* como efecto lateral de la definición de un "tipo estructurado" mediante CREATE TYPE (vea posteriormente en esta sección); por lo tanto, ignoraremos por ahora a REF. En lo que se refiere a los tipos ARRAY y ROW, en realidad no pueden ser *definidos* como tales (no hay declaraciones CREATE ARRAY TYPE o CREATE ROW TYPE); sólo pueden ser *usados* llamando al generador de tipos relevante "en línea" dentro de (por ejemplo) una instrucción CREATE TABLE. Ésta podría ser un ejemplo del uso del **tipo ARRAY:**

```
CREATE TABLE VENTAS (
     ART# CHAR(5),
       CANT INTEGER ARRAY [12],
     PRIMARY KEY ( ART# ) ) ;
```

Aquí la columna CANT tiene valor de arreglo; un valor CANT consiste en un arreglo de 12 elementos, cada uno de tipo INTEGER. *Nota:* Los arreglos de SQL están limitados a una dimensión, y no está permitido que los elementos sean arreglos.

Éste es un ejemplo de una consulta contra VENTAS:

```
SELECT ART#
FROM  VENTAS
WHERE CANT [3] > 100 ;
```

("Obtener los números de artículo para los artículos cuyas ventas de marzo exceden de 100"; observe la referencia en subíndice.) Y éste es un ejemplo de la inserción de una fila:

```
INSERT INTO VENTAS ( ART#, CANT )
VALUES ( 'X4320',
         ARRAY [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ) ;
```

(observe la literal tipo arreglo).

Los **tipos ROW** son similares. Por ejemplo:

```
CREATE TABLE CLIEN
     ( CLIEN# CHAR(3),
     DOM ROW ( CALLE  CHAR(50),
     CIUDAD CHAR(25), ESTADO CHAR(2),
     CP    CHAR(5) ), PRIMARY KEY (
     CLIEN# ) ) ;
```

Consulta de ejemplo:

```
SELECT CLIEN#
FROM  CLIEN
WHERE DOM. ESTADO = CA' ;
```

Inserción de ejemplo:

```
INSERT INTO CLIEN ( CLIEN#, DOM )
VALUES ( '001', ROW ( '1600 Pennsylvania Ave.',
                      'Washington', 'DC, '20500'
```

Tipos DISTINCT

La nueva instrucción CREATE TYPE crea un **tipo definido por el usuario:** ya sea un tipo *DISTINCT* o un tipo *"estructurado"* (observe de paso que los tipos *generados* —vea la subsección anterior— no son considerados como tipos "definidos por el usuario" en el mismo sentido). En esta subsección tratamos solamente a los tipos DISTINCT. Un tipo DISTINCT (ponemos "DISTINCT" en mayúsculas para enfatizar el punto de que la palabra no está siendo usada en su sentido usual del lenguaje natural) es un caso especial limitado de un tipo definido por el usuario. En particular, su implementación física debe involucrar exactamente uno de los tipos escalares integrados. Ésta es la sintaxis para la definición de un **tipo DISTINCT:**

```
CREATE TYPE <nombre de tipo>
    AS <nombre de tipo escalar integrado> FINAL
      [<opciones de conversión de tipo> ]
      [<lista de especificaciones de método separadas con comas> ] ;
```

Este es un ejemplo:

```
CREATE TYPE PESO AS NUMERIC (5,1) FINAL ;
```

Explicación:

1. El tipo PESO hereda los operadores de comparación que se aplican al tipo subyacente (es decir, el tipo NUMERIC). Sin embargo, observe que los valores de PESO sólo son com parables entre *sí y con ningún otro.* Por lo tanto, por ejemplo, si PS es una variable SQL de tipo PESO la siguiente comparación no es válida:

   ```
   PS > 14.7
   ```

   Sin embargo, si por el contrario suponemos que las <*opciones de conversión de tipo*> adecuadas han sido especificadas (aquí omitimos los detalles), las siguientes comparaciones serían válidas:

   ```
   PS > CAST ( 14.7 AS PESO ) CAST

   ( PS AS NUMERIC ) > 14.7
   ```

   Además, las dos invocaciones a CAST que mostramos, podrían ser abreviadas como PESO(14.7) y NUMERIC(PS), respectivamente. *Nota:* Aquí, los nombres de función PESO y NUMERIC son especificados por el definidor de tipo (en las <*opciones de conversión de tipo*>)\ *no* están implicados por el nombre del tipo que está siendo definido o el nombre de su tipo subyacente.
2. Comentarios similares se aplican a la asignación; es decir, un valor PESO sólo puede ser asig nado a un destino de tipo PESO (y ninguna otra cosa puede ser asignada a dicho destino).
3. El tipo PESO no hereda automáticamente otros operadores del tipo subyacente. Sin embargo, la <*especifwación de método*» (el ejemplo no muestra alguna) permite que el definidor de tipo especifique "métodos" —vea la siguiente subsección— que se aplican a valores y variables

de tipo PESO. Los procedimientos y funciones normales de SQL también pueden ser definidos para que operen sobre valores y variables de tipo PESO. Por ejemplo, podríamos definir una función llamada SUMAP que sume dos pesos para obtener un tercero, lo que permitiría, por lo tanto, que el usuario escribiera expresiones tales como las siguientes:

```
SUMAP ( PS1, PS2 )

SUMAP ( PS1, PESO ( 14.7 ) )
```

4. Debe aparecer la especificación FINAL. Vea la subsección que sigue a continuación.

Tipos estructurados

La otra clase de tipo definido por el usuario es un **tipo estructurado.** Estos son un par de ejemplos:

```
CREATE TYPE PUNTO AS ( X FLOAT, Y FLOAT ) FINAL ;

CREATE TYPE SEGLINEA AS ( INICIO PUNTO, FIN PUNTO ) FINAL ;
```

Explicación:

1. Decimos que el tipo PUNTO tiene *atributos* X y Y (no lo confunda con los atributos de tupia y de relación, tal como los definimos en la parte II de este libro); en forma similar, decimos que el tipo SEGLINEA tiene atributos INICIO y FIN. Un atributo puede ser de cualquier tipo conocido.
2. Por desgracia, los atributos mencionados en la definición de un tipo estructurado consti tuyen la *implementación física* de los valores del tipo en cuestión, y no una "representación posible" en el sentido del capítulo 5. Por lo tanto, los puntos del ejemplo están implementados físicamente en términos de sus coordenadas cartesianas.
3. Cada definición de atributo produce automáticamente la definición de un operador *observador* (por lo general, "obtener") y un operador *matador* (por lo general, "colocar"), para los cuales se usa la sintaxis de calificación por punto.* Por ejemplo, sean Z, P y SL variables SQL de tipos FLOAT, PUNTO y SEGLINEA, respectivamente. Entonces, lo siguiente es válido:

```
p.X         /* obtiene el valor del componente x del punto P */

SL.INICIO.X         /* obtiene el valor del componente X del */
                       /* punto INICIO del segmento de linea SL */

SET P.X = Z ;               /* pone el componente X del punto P */
                            /♦ al valor de Z                     */

SET SL. INICIO.X = Z ;  /* pone el componente X del punto INICIO */
                        /* del segmento de línea SL               */
                        /* al valor de Z                          */
```

*Debemos observar que el "mutador" no es verdaderamente un mutador en el sentido del capítulo 5 —es decir, un operador de actualización— debido a que está definido para que regrese un valor. Las consecuencias (desagradables) de este hecho están desafortunadamente fuera del alcance de este breve apéndice. Vea la referencia [3.3] para una explicación más detallada.

4. Los otros únicos operadores disponibles para estos tipos —debido a que no hemos definido ninguno adicional— son las comparaciones de igualdad y los operadores de asignación, que están disponibles para *todo* tipo. Observe en particular, que los "selectores" PUNTO y SEGLINEA (en el sentido del capítulo 5) no están definidos automáticamente y por lo tanto, no hay literales PUNTO y SEGLINEA.
5. La especificación FINAL significa que fallará cualquier intento para definir otro tipo como un subtipo propio de cualquiera de estos tipos (vea la sección B.3); es decir, estos tipos son tipos hoja y así permanecerán.
6. ¿Están los tipos PUNTO y SEGLINEA (o los tipos estructurados en general) "encapsulados"? Desgraciadamente, la respuesta a esta pregunta parece depender del contexto. Por ejemplo, cuando un tipo estructurado es usado como tipo para alguna *columna,* la respuesta es *sí* (algo así). Sin embargo, cuando es usado como tipo para alguna *tabla* —vea la sec ción B.4— la respuesta es definitivamente *no.*

   Nota: La razón por la cual decimos "algo así" en el primer caso se debe a que incluso en ese caso, los operadores de "obtener y poner" exponen en efecto los atributos del tipo (como ya dijimos) y esos operadores no pueden ser anulados. Tal vez debiéramos usar el término *"seudoencapsulados"* —o tal vez **seudoescalar**— para referirnos al primer caso.

Entonces, ésta es la sintaxis general para la definición de un tipo estructurado que no es un subtipo propio (vea la sección B.3 para una explicación del caso del subtipo propio):

```
CREATE TYPE <nombre de tipo>
    AS  ( <lista de definiciones de atributo separadas con comas>  )
        [  <implementación de tipo de referencia>  J [
     [  NOT  ]  INSTANTIABLE  ]
        [  NOT  ]  FINAL [  <lista de especificaciones de método separadas
     con comas>  ]   ;
```

Explicación:

1. La <lista de definiciones de atributo separadas con comas> no debe estar vacía.
2. Explicamos la <*implementación de tipo de referencia*> opcional en la sección B.4.
3. NOT INSTANTIABLE significa que el tipo es un tipo *ficticio* en el sentido del capítulo 19. Lo predeterminado es INSTANTIABLE.
4. NOT FINAL significa que el tipo puede tener subtipos propios. FINAL significa que no puede.
5. Los operadores que se aplican a los valores y variables de un tipo *T*estructurado dado son:
   a. Los observadores y mutadores de atributos descritos anteriormente.
   b. La asignación "=", y posiblemente "<" (estos últimos están definidos por medio de una declaración independiente CREATE ORDERING FOR *T,* aunque no queda claro por qué es necesario crear un ordenamiento sólo para definir "=").
   c. Los procedimientos y funciones que están definidos para que tomen un parámetro de tipo *T* (o cualquier supertipo propio; vea la sección B.3).
   d. Los *métodos* que están definidos para tomar un parámetro de "destino" especial de tipo *T* o cualquier supertipo propio (de nuevo, vea la sección B.3). *Nota:* Aquí "métodos" son los métodos en el sentido tradicional de objetos (vea el capítulo 24); es decir, son operadores

que tratan a un parámetro como especial. Si el método *M* está definido para el tipo *T*, y *X* es una expresión de tipo *T*, entonces se usa la sintaxis especial de calificación por punto *X.M* para llamar a *M* sobre *X*. (Aquí suponemos, por razones de simplicidad, que *M* no toma otros parámetros.)

La *Especificación de método*> define las "signaturas de especificación" de método en el sentido de los capítulos 19 y 24. Sin embargo, también incluye muchas otras cosas, muchas de naturaleza de implementación y (en mi opinión) fuera de lugar. El código real que implementa al método se define en cualquier otro lugar.

B.3 HERENCIA DE TIPO

El soporte de SQL3 para la herencia de tipo no es muy ortogonal, ya que sólo se aplica a los tipos estructurados; es decir, no soporta la herencia de tipo para los tipos integrados, generados* o DISTINCT. Además, no soporta en absoluto la herencia múltiple. En esta sección consideramos la herencia de tipo sólo para tipos estructurados "seudoescalares" (que pueden ser considerados como el caso aceptable, más o menos); dejamos el caso de los tipos estructurados *no encapsulados* para la sección B.5 posterior.

Las diferencias lógicas más grandes entre el modelo de herencia de tipo, que presentamos en el capítulo 19, y la herencia de tipo "seudoescalar" de SQL3 son las siguientes:

- SQL3 soporta la herencia estructural y también de comportamiento.
- SQL3 no distingue adecuadamente entre valores y variables; por lo tanto, en particular, no distingue entre sustituibilidad de valor y de variable.
- SQL3 no tiene soporte para restricciones de tipo, y por lo tanto, tampoco para la especialización por restricción.
- SQL3 requiere que los operadores de actualización ("imitadores") se hereden en forma in condicional.

Una consecuencia de estas diferencias es que SQL3 permite "círculos no circulares" y cosas sin sentido similares, como dijimos en el capítulo 19. Otra es que determinadas comparaciones de SQL3 darán *falso* cuando debieran dar *verdadero* (debido a que, por ejemplo, en SQL3 un valor de tipo más específico ELIPSE puede tener semiejes iguales y por lo tanto, corresponder a un círculo en la realidad).

Así es como podemos ver nuestro ejemplo usual de elipses y círculos en SQL3:

```
CREATE TYPE ELIPSE
  AS ( A LONGITUD, B LONGITUD, CTRO PUNTO ) NOT
       FINAL ;

CREATE TYPE CIRCULO UNDER ELIPSE
  AS ( R LONGITUD )              — no realista (vea más adelante)
       NOT FINAL ;
```

*Con excepción, posiblemente, de tipos de fila. Hay algunas confusiones respecto de la herencia de tipo de fila, pero los detalles están fuera del alcance de este apéndice.

Sin embargo, observe que con estas definiciones, la implementación física del tipo CIRCULO involucrará cuatro componentes: A, B y CTRO (heredados del tipo ELIPSE) y R (especificado sólo para el tipo CIRCULO). Por supuesto, para cualquier círculo dado, tres de estos componentes tendrán el mismo valor. En forma alterna, podríamos definir un *método* R para el tipo CIRCULO (en lugar del atributo R); el tipo CIRCULO tendría entonces la misma implementación física que el tipo ELIPSE, y esa implementación involucraría entonces una menor redundancia. Por otro lado, si C es una variable de tipo declarado CIRCULO, entonces la asignación a C.R sería válida con el primero de estos diseños pero no con el segundo. Luego, de nuevo, la asignación a C.R (en caso que estuviera soportada) produciría —en general— ¡un "círculo" tal que C.R # C.A!

Éstos son algunos puntos más de diferencias entre el modelo de herencia de tipo, que describimos en el capítulo 19, y la herencia de tipo tal como es soportada en SQL3:

- SQL3 usa el término "directo" (como en *subtipo directo)* en lugar de "inmediato", que es el más adecuado.
- SQL3 usa el término "supertipo maximal" en lugar de "tipo raíz".
- SQL3 soporta un operador llamado TREAT (similar a TREAT DOWN). Por ejemplo, la analogía de SQL3 para la expresión TREAT_DOWN_AS_CIRCULO(E) es TREAT E AS CIRCULO.
- SQL3 también soporta un operador que podría ser visto (aproximadamente) como "TREAT UP". Por ejemplo, considere la siguiente llamada de operador:

  ```
  AREA ( C AS ELIPSE )
  ```

  La especificación "AS ELIPSE" obliga a que sea llamada la versión *ELIPSE* del operador AREA, aunque el tipo declarado de la variable C sea CIRCULO. *Nota:* Este operador puede ser usado solamente en determinados contextos; para ser específicos, sólo puede ser usado para "tratar hacia arriba" el argumento de una llamada a algún operador definido por el usuario, como en el ejemplo. Hacemos notar que esta funcionalidad parece sugerir alguna confusión sobre las cuestiones del modelo contra la implementación; después de todo, los usuarios ni siquiera tienen que saber que hay dos versiones de AREA.
- SQL3 también soporta un método de la forma XSPECIFICTYPE, el cual regresa el tipo más específico de su argumento *X* como una cadena de caracteres.
- Las analogías de SQL3 para *IS_T(X)* e *IS_MS_T(X)* se ven de esta forma:

  ```
  x IS OF ( T )
  X IS OF ( ONLY T )
  ```

B.4 TIPOS DE REFERENCIA

Recuerde lo que dijimos en el capítulo 24 en el sentido de que los sistemas de objetos involucran sólo una buena idea, es decir, el *soporte apropiado a los tipos de datos* (o dos buenas ideas, si contamos por separado la herencia de tipo). Como hemos visto, SQL3 incluye algo de esta "buena" funcionalidad (aunque su soporte está muy lejos del ideal). Sin embargo, por desgracia, también incluye cierta funcionalidad relacionada que es —en nuestra opinión— muy mala. En realidad, SQL3 está muy cerca de cometer *el primer gran error garrafal* (al igualar tablas y clases) y tam-

bien *el segundo gran error garrafal* (al mezclar apuntadores y tablas). Y hay que decir que la "justificación" para hacerlo no es muy clara, al menos para mí; parece ser algo más que una idea vaga que las características en cuestión hacen que SQL3 sea en cierta forma más "parecido a objetos". Sea como fuere, tratemos ahora de explicar cuáles son de hecho las características importantes. En primer lugar, SQL3 permite definir una tabla base no sólo en términos de un conjunto explícito de columnas de tipo nombradas en la forma usual, sino también en términos de un *tipo estructurado definido por el usuario*. Por ejemplo:

```
CREATE TYPE TIP0_DEPT0
  AS ( DEPT0# CHAR(3),
       NOMDEPTO CHAR(25),
       PRESUPUESTO MONEY )
       REF IS SYSTEM GENERATED ... ;

CREATE TABLE DEPTO OF TIP0_DEPT0
     ( REF IS ID_DEPTO SYSTEM GENERATED,
       PRIMARY KEY ( DEPTO# ) , UNIQUE (
       IDDEPTO ) ) . . . ;
```

Explicación:

1. Dada la definición del tipo estructurado *T*, el sistema genera automáticamente un *tipo de referencia* asociado ("tipo REF") llamado REF(T). Los valores de tipo REF(7) son "refe rencias" a filas de tipo *T* dentro de alguna tabla base* que ha sido definida como "OF" tipo *T* (vea el punto 3 más adelante). *Nota:* Por supuesto, *T* puede ser usado en otros contextos; por ejemplo, como el tipo declarado de alguna variable *V* o alguna columna C. Sin embargo, ningún valor REF(7) está asociado con esos otros usos.
2. REF IS SYSTEM GENERATED en una declaración CREATE TYPE significa que los va lores reales del tipo REF asociado, son proporcionados por el sistema (otras opciones —por ejemplo, REF IS USER GENERATED— están disponibles, pero aquí omitimos los de talles). Lo predeterminado es REF IS SYSTEM GENERATED.
3. Una tabla base puede ser definida (por medio de CREATE TABLE) para ser "OF" algún tipo estructurado. Sin embargo, aquí la palabra reservada OF no es en realidad muy adecua da, debido a que la tabla en realidad *no* es "OF" (del) tipo en cuestión, así como tampoco ninguna de sus filas.[+] En particular, es posible especificar columnas adicionales por encima de las "columnas" (o atributos, más bien) del propio tipo estructurado. En realidad, *debe* es pecificarse al menos una de tales columnas adicionales —es decir, una columna del tipo REF aplicable— aunque la sintaxis para la definición de esa columna no es la sintaxis usual de definición de columna, sino que en su lugar se ve de esta forma:

   ```
   REF IS <nombre de columna> SISTEM GENERATED
   ```

   Esta columna adicional es usada para contener IDs ("referencias") únicas para las filas de la tabla base en cuestión (está implícita la especificación UNIQUE *(<nombre de columna>)*, aunque también puede ser especificada explícitamente como en nuestro ejemplo). El ID para

---

*O posiblemente alguna vista. Los detalles del caso de vistas están fuera del alcance de este apéndice. [t]Por lo tanto, observe en particular que si el tipo declarado de algún parámetro *P* para algún operador *Op* es algún tipo estructurado *T*, una fila de una tabla base que ha sido definida para que sea "OF" tipo *T*, no puede ser aprobada como argumento correspondiente para una invocación de ese operador *Op*.

una fila dada es asignado cuando la fila es insertada y permanece asociado con esa fila* hasta que es eliminada. *Nota:* La repetición de la especificación SYSTEM GENERATED aparentemente es requerida. (Algo sorprendente, una columna SYSTEM GENERATED *puede* ser una columna de destino en una operación INSERT o UPDATE, aunque se aplican consideraciones especiales; aquí omitimos los detalles.) Además, en primer lugar no está claro por qué debe ser necesario definir que la tabla sea "OF" algún tipo estructurado —en lugar de sólo definir una columna adecuada de la forma usual— para obtener la funcionalidad de "ID único".

4. Como indicamos en la sección B.2, un tipo estructurado como TIPO_DEPTO no se consi dera como encapsulado cuando es usado como base para la definición de una tabla base, aunque *sí* es considerado así (más o menos) en otros contextos. Por lo tanto, la tabla base DEPTO del ejemplo tiene cuatro columnas, y no sólo dos como sería si estuviera encapsu lado TIPO_DEPTO.
5. Hemos mostrado a DEPTO# como clave primaria para la tabla DEPTO. Sin embargo, tomando en cuenta que las filas de DEPTO tienen IDs ("referencias") únicas, podríamos usar esas IDs como valores de clave primaria, si lo deseáramos, así:

```
CREATE TABLE DEPTO OF TIP0_DEPT0
     ( REF IS ID_DEPTO SYSTEM GENERATED,
       PRIMARY KEY ( IDDEPTO ), UNIQUE (
       DEPT# ) ) . . . ;
```

Extendamos ahora el ejemplo para presentar una tabla base EMP, así:

```
CREATE TABLE EMP
     ( EMP#    CHAR(5),
       NOMEMP  CHAR(25),
       SALARIO MONEY,
     ID_DEPTO REF ( TIPODEPTO ) SCOPE DEPTO
     REFERENCES ARE CHECKED ON DELETE CASCADE,
     PRIMARY KEY  ( EMP# ) ) . . . ;
```

Normalmente, la tabla base EMP incluiría una columna de clave externa DEPTO# para hacer referencia a los departamentos por número de departamento. Sin embargo, aquí tenemos una columna de "referencia" ID_DEPTO —que no está declarada explícitamente para ser una columna de clave externa como tal— que, en su lugar, hace referencia a los departamentos mediante sus "referencias". SCOPE DEPTO especifica la tabla referida aplicable. REFERENCES ARE CHECKED significa que se mantendrá la integridad referencial (REFERENCES ARE NOT CHECKED permitiría "referencias colgadas"; no está claro por qué sería necesario especificar esta opción). ON DELETE... especifica una regla de eliminación, similar a las reglas de eliminación de clave externa usuales (están soportadas las mismas opciones); pero observe que en realidad si resulta que la columna ID_DEPTO de la tabla base DEPTO no puede ser actualizada (vea el punto 3 anterior), entonces no es necesaria una regla ON UPDATE correspondiente.

Ahora consideraremos algunas consultas de ejemplo que involucran a esta base de datos. Aquí está primero una formulación SQL3 "obtener el número de departamento del empleado El":

```
SELECT EX.IDDEPTO -> DEPT0# AS DEPTO*
FROM  EMP EX
WHERE   EX.EMP# = 'E1'  ;
```

*Aquí hay algo de circularidad: "esa fila" sólo puede significar "la fila que tiene el ID particular en cuestión". ¡Observe la confusión de valor contra variable!

Observe la operación de **desreferencia*** en la cláusula SELECT (la expresión EX.ID_DEPTO -» DEPTO# regresa el valor DEPTO# de la fila DEPTO a la que apunta el valor ID_DEPTO en cuestión). Observe también que la especificación AS en esa cláusula es más o menos requerida (si se omitiera, la columna resultante podría tener un nombre "dependiente de la implementación"). Observe por último la naturaleza contraintuitiva de la cláusula FROM; el valor DEPTO# que va a ser recuperado viene de DEPTO y no de EMP, pero los valores de ID_DEPTO vienen de EMP y no de DEPTO. *Nota:* Es difícil resistir la tentación de señalar que esta consulta probablemente se desempeñará *peor* que su contraparte convencional de SQL (la cual accedería sólo a una tabla y no a dos). Hacemos esta observación debido a que el argumento usual a favor de las "referencias" es que supuestamente mejoran el rendimiento.

A propósito, si la consulta hubiera sido "obtener el *departamento* (en lugar de sólo el número de departamento) para el empleado El", la operación de desreferencia habría sido algo diferente:

```
SELECT DEREF ( EX.ID_DEPTO ) AS DEPTO
FROM   EMP EX
WHERE    EX.EMP# ■ 'E1 '  ;
```

Además, la llamada a DEREF que mostramos habría regresado no un valor *de fila,* sino un valor (escalar) "encapsulado".

Éste es otro ejemplo: "obtener los números de empleado para los empleados que están en el departamento DI" (observe que aquí la desreferencia está en la cláusula WHERE):

```
SELECT EX.EMP#
FROM   EMP EX
WHERE   EX.IDDEPTO -> DEPTO# = 'D1' ;
```

Y éste es un ejemplo de INSERT (inserción de un empleado):

```
INSERT INTO EMP ( EMP#, IDDEPTO )
       VALUES ( 'E5[1], ( SELECT DX.IDDEPTO FROM   DEPTO
                          DX WHERE  DX. DEPTO* = ' D2' ) )
                          ;
```

B.5 SUBTABLAS Y SUPERTABLAS

Considere los siguientes tipos estructurados:

```
CREATE TYPE TIPOEMP
  AS ( EMP# . . . , DEPT0# . . . )
     REF IS SYSTEM GENERATED
     NOT FINAL ... ;

CREATE TYPE TIPOPGMR UNDER TIP0_EMP
  AS ( LANG ... ) NOT FINAL ... ;
```

A propósito, observe que TIPO_PGMR no tiene cláusula REF IS...; en vez de ello, hereda en efecto una de estas cláusulas de su supertipo inmediato ("directo") TIPO_EMP.

---

*La mayoría de los lenguajes que soportan la desreferencia también soportan una operación de *referencia,* pero SQL3 no lo hace.

Ahora considere las siguientes definiciones de tablas base:

```
CREATE TABLE EMP OF TIPO_EMP
     ( REF IS IDEMP SYSTEM GENERATED,
       PRIMARY KEY ( EMP# ), UNIQUE (
       ID_EMP ) ) ;

CREATE TABLE PGMR OF TIP0_PGMR UNDER EMP ;
```

Observe la especificación UNDER EMP en la definición de la tabla base PGMR (observe también la omisión de las especificaciones REF IS..., PRIMARY KEY y UNIQUE para esa tabla base). Decimos que las tablas base PGMR y EMP son, respectivamente, una **subtabla y** la **supertabla** inmediata ("directa") correspondiente; PGMR hereda las columnas (etcétera) de EMP y añade una columna LANG propia de ella. La intuición detrás de este ejemplo es que quienes no son programadores sólo tienen una fila en la tabla EMP, mientras que los programadores tienen una fila en ambas tablas; así, toda fila en PGMR tiene una contraparte en EMP, aunque lo contrario no es cierto.

Las operaciones de manipulación de datos sobre estas tablas se comportan de la manera siguiente:

- *SELECT.* SELECT en EMP se comporta normalmente. SELECT en PGMR se comporta como si PGMR incluyera de hecho las columnas de EMP y también la columna LANG. *Nota:* El calificador ONLY puede ser usado en una <*referencia de tabla*> para excluir filas que tienen contrapartes en alguna subtabla de la tabla en cuestión. Por lo tanto la operación SELECT...FROM ONLY EMP, por ejemplo, se comporta como si EMP incluyera sólo filas que no tienen contraparte en PGMR.
- *INSERT.* INSERT en EMP se comporta normalmente. INSERT en PGMR causa, en efecto, que aparezcan nuevas filas en EMP y PGMR.
- *DELETE.* DELETE en EMP causa que desaparezcan filas de EMP y (cuando las filas en cuestión corresponden a programadores) también de PGMR. DELETE en PGMR causa que desaparezcan filas de EMP y de PGMR.
- *UPDATE.* La actualización de LANG —por medio de PGMR, necesariamente— actualiza sólo a PGMR; la actualización de las demás columnas —por medio de EMP o PGMR— actualiza ambas tablas (de manera conceptual).

Observe las siguientes implicaciones de lo anterior:

- Suponga que el empleado Joe se convierte en programador. Si tratamos simplemente de in sertar una fila para Joe en PGMR, el sistema tratará de insertar una fila para Joe también en EMP; un intento que fallará, por supuesto. En vez de ello, tenemos que eliminar la fila de Joe que está en EMP y luego insertar una fila adecuada en PGMR.
- En forma alterna, suponga que el empleado Joe deja de ser un programador. Esta vez tenemos que eliminar la fila de Joe, ya sea de EMP o de PGMR (sin importar la tabla que especi fiquemos, el efecto será eliminarla en ambas), y luego insertar una fila adecuada en EMP.

¿Qué tiene que ver todo esto con la herencia de tipo? Por lo que podemos ver, la respuesta es *nada;* es decir, nada salvo que (por razones que no son muy claras, por no decir más) SQL3 requiere que una subtabla y su supertabla inmediata ("directa") sean definidas sobre tipos estructurados que son, respectivamente, un subtipo y su supertipo inmediato ("directo"). Observe

que no hay sustituibilidad involucrada en este acuerdo; también observe que los tipos involucrados definitivamente no están "encapsulados".

Entonces, ¿qué nos dan las subtablas y supertablas? La respuesta parece ser "muy poco", al menos en el nivel del *modelo** Es cierto que podemos obtener algunos ahorros de *implementation,* si la subtabla y su supertabla están almacenadas físicamente como una sola tabla en el disco; pero, por supuesto, no deberíamos permitir que tales consideraciones tuvieran algún efecto en el modelo como tal. En otras palabras, no sólo no queda claro (como indicamos en el párrafo anterior) por qué las "sub y super" tablas tienen que apoyarse en "sub y super" tipos estructurados, sino que tampoco queda muy claro por qué están soportadas las características *del todo.*

B.6 OTRAS CARACTERÍSTICAS

CREATE TABLE

SQL3 soporta una **opción LIKE** en CREATE TABLE, que permite que algunas o todas las definiciones de columna de una nueva tabla base sean copiadas a partir de alguna tabla nombrada existente (observe el *nombrada;* no es posible especificar una expresión de tabla cualquiera en lugar de un nombre de tabla).

Expresiones de tabla

En el capítulo 5 describimos una cláusula **WITH,** cuyo propósito era introducir nombres abreviados para determinadas expresiones. SQL3 incluye una construcción similar, pero su uso está limitado a sólo las expresiones de *tabla.* Éste es un ejemplo:

```
WITH EMPS_MUCHA_ANTIG AS ( SELECT * FROM  EMP WHERE
   FECHACONTRATADO < DATE '1980-01-01' )

SELECT EMP#, { CT_MUCHA_ANTIG - 1 ) AS #_DE_CONTEMP_EMPS_MUCHA_ANTIG
FROM  EMPS_MUCHA_ANTIG AS L1 ,
     ( SELECT DEPT0#, COUNT(*) AS CT_MUCHA_ANTIG
       FROM  EMPS_MUCHA_ANTIG
       GROUP BY DEPTO# ) AS L2
WHERE  L1.DEPTO* = L2.DEPT0# ;
```

("Para cada empleado que ha estado en la compañía desde 1979 o antes, obtener el número de empleado y la cuenta de los demás empleados con esta característica que estén en el mismo departamento".)

La cláusula WITH de SQL3 puede ser particularmente usada para formular determinadas consultas **recursivas.** Por ejemplo, dada la tabla PADRE_DE (padre, hijo), la siguiente consulta recursiva regresa todos los pares de personas *(a,b)* tales que *a* es el ancestro de *b.* Observe que la definición del nombre ANCESTRO_DE incluye una referencia a ANCESTRO_DE sí mismo.

---

*Tal vez debemos recordarle nuestro propio enfoque preferido para este aspecto —si en realidad es este aspecto— el cual utiliza *vistas* (vea el ejemplo al final de la sección 13.5).

```
WITH RECURSIVE ANCESTRODE ( ANCESTRO, DESCENDIENTE )
AS ( SELECT PADRE, HIJO
     FROM  PADRE_DE
     UNION
     SELECT A.PADRE, P.HIJO
     FROM  ANCESTROJJE AS A, PADREDE AS P
     WHERE A.HIJO " P.PADRE )

SELECT *
FROM ANCESTRO DE  ;
```

Expresiones condicionales

SQL3 proporciona una nueva condición **DISTINCT** (no la confunda con la condición UNIQUE existente, vea el apéndice A) para probar si dos filas son "distintas". Sean las filas en cuestión *Izquierda y Derecha; Izquierda* y *Derecha* deben contener la misma cantidad de componentes, digamos *n.* Sean *Ii* los ¡esitnos componentes de *Izquierda* y *Di* los de *Derecha, (i* = 1, 2,..., n). El tipo de ***Ii*** debe ser compatible con el tipo de *Di.* Entonces la expresión

```
Izquierda IS DISTINCT FROM Derecha
```

regresa/a/so si y sólo si, para todos los *i,* (a) ***"Ii*** = £>/" es *verdadero* o (b) *Ii* y *Di* ambos son nulos; en caso contrario regresa *verdadero.* (En otras palabras, *Izquierda* y *Derecha* son "distintas" si y sólo si no son "duplicadas" entre ellas en el sentido del capítulo 18.) Observe que una condición DISTINCT nunca da como resultado *desconocido.*

SQL3 también proporciona una nueva condición **SIMILAR,** la cual (al igual que LIKE) está orientada hacia la coincidencia de patrones de cadenas de caracteres; es decir, a probar una cadena de caracteres dada para ver si se apega a algún patrón prescrito. La diferencia es que SIMILAR soporta un rango de posibilidades mucho más amplio ("comodines", etcétera) que el que tiene LIKE. La sintaxis es:

```
<expresión de cadena de caracteres>     NOT  ]  SIMILAR TO <patrón> [
                                                ESCAPE <escape> ]
```

Las especificaciones <*patrón*> y <*escape*> son esencialmente las mismas que las de LIKE, con excepción de que <*patrón*> puede involucrar caracteres especiales adicionales; no sólo "_" y "%" como LIKE, sino también "*", "+", "-" y muchos otros. La pretensión general parece ser la de dar soporte al análisis sintáctico de expresiones escritas en algún lenguaje formal. *Nota:* Vale la pena mencionar que las reglas para SIMILAR fueron copiadas de un operador similar en POSIX.

Al cierre de esta sección hacemos notar que, tomando en cuenta la existencia del nuevo tipo integrado BOOLEAN, las expresiones condicionales ahora son en realidad sólo un tipo especial de expresión escalar (como debieran haber sido desde hace mucho).

Integridad

SQL3 soporta **RESTRICT** <*acción referential*, que es similar pero no idéntico, a NO ACTION (vea el capítulo 8 para una explicación de la diferencia). También incluye algún soporte para los **procedimientos disparados** ("disparadores"); en particular, incluye una declaración CREATE

TRIGGER, la cual define a un *disparador;* es decir, una combinación de una especificación de *evento* y una especificación de *acción,* donde:

- Un *evento* es un INSERT, UPDATE (opcional sobre columnas especificadas) o DELETE en contra de una tabla nombrada específica.
- La *acción* es una acción (de hecho, un procedimiento) a realizarse después (AFTER) o antes (BEFORE) de que suceda el evento especificado.

Para ser más precisos, la *acción* consiste en una expresión condicional opcional (con un valor predeterminado de TRUE), más un procedimiento SQL que será ejecutado si y sólo si la condición es *verdadera* cuando sucede el *evento.* El usuario puede especificar si la acción se realiza sólo una vez por ocurrencia del evento o una vez por cada columna (FOR EACH ROW) de la tabla con la cual está asociado el evento. Además, la especificación de la acción puede hacer referencia a valores "antes" y "después" dentro de la tabla asociada con el evento especificado, proporcionando así un nivel primitivo de soporte para *restricciones de transición* (entre otras cosas).

Actualización de vistas

SQL3 extiende el soporte de SQL para la **actualización de vistas,** para incluir vistas "UNION ALL" y vistas de junta de uno a uno y de uno a muchos. Extensiones similares se aplican a los cursores.

Administración de transacciones

SQL3 incluye varias características nuevas de administración de transacciones:

- Una instrucción **START TRANSACTION** explícita (con los mismos operandos que SET TRANSACTION; vea el capítulo 14).
- Una opción **WITH HOLD** en DECLARE CURSOR (de nuevo, vea el capítulo 14).
- Soporte para **puntos de resguardo** (vea el comentario a la referencia [14.11]).

Seguridad

SQL3 soporta **privilegios SELECT específicos de columna** (el privilegio "SELECT(JC)" permite que su poseedor haga referencia a una columna *x* específica, de una tabla nombrada específica, dentro de una *<expresión de tabla>).* También soporta **papeles** definidos por el usuario; un ejemplo podría ser GCONT, que significa a todos los del departamento de contabilidad. Una vez que ha sido creado, al papel se le pueden otorgar privilegios como si fuera un ID de usuario. Además, los papeles pueden ser otorgados igual que los privilegios, e igual que todos los privilegios, pueden ser otorgados a un ID de usuario o a otro papel.

Información íaltante

Sólo haremos una observación bajo este encabezado; es decir, que las nuevas facilidades de tipo de SQL3 son (en forma bastante predecible) complicadas por la presencia de nulos. Por ejemplo,

sea *V* una variable de algún tipo estructurado *T.* Entonces es posible que algún componente de *V* pueda ser nulo (en cuyo caso la expresión condicional *V = V* daría *desconocido),* e incluso la expresión condicional VIS NULL ¡daría *falso]* De hecho, podemos decir, en general, que si *((V= V)* IS NOT TRUE) IS TRUE es *verdadero,* entonces *Ves* nulo o incluye un componente nulo.

Apoyo para la toma de decisiones

SQL3 incluye apoyo para las opciones **GROUPING SETS, ROLLUP y CUBE** en GROUP BY, tal como lo describimos en el capítulo 21.

APÉNDICE C

Abreviaturas, acrónimos y símbolos

ACID	atomicidad/consistencia/aislamiento/durabilidad
ACM	Asociación para las Máquinas de Cómputo
ADT	tipo de dato abstracto
ANSI	Instituto Nacional Americano de Estándares
ANSI/SPARC	literalmente, ANSI/Comité de Planeación y Requerimientos de Sistemas; se usa para referirse a la arquitectura de sistemas de bases de datos de tres niveles descrita en el capítulo 2
ARIES	algoritmo para la recuperación y aislamiento explotando la semántica
BB	lo mismo que GB
BCS	Sociedad Británica de Computadoras
BLOB	objeto binario grande
BNF	forma Backus-Naur o forma normal Backus
CACM	*Communications of the ACM* (publicación de la ACM)
CAD/CAM	diseño asistido por computadora/manufactura asistida por computadora
CASE	ingeniería de software asistida por computadora
CDO	objeto de definición de clase
CIM	fabricación integrada con computadora
[1]CLI	interfaz a nivel de llamada (parte del estándar SQL)
CLOB	objeto de caracteres grande
CNF	forma normal conjuntiva
CODASYL	literalmente, Conferencia sobre Lenguajes de Sistemas de Datos; se usa para referirse a determinados sistemas prerrelacionales (de red) tales como IDMS
CPU	unidad central de procesamiento
CS	estabilidad de cursor (DB2)
CWA	Suposición del Mundo Cerrado
DA	administrador de datos
DBA	administrador de base de datos

DB/DC	base de datos/comunicaciones de datos
DBMS	sistema de administración de base de datos
DBP&D	*Database Programming & Design* (revista, ahora está en línea)
DBTG	literalmente, Grupo de Trabajo de Base de Datos; se usa en forma indistinta con CODASYL (en contextos de base de datos)
DC	comunicaciones de datos
DCO	"omitir revisión de dominio"
DDB	base de datos distribuida
DDBMS	DBMS distribuido
DDL	lenguaje de definición de datos
DES	Estándar de Cifrado de Datos
DJ	dependencia de junta
DML	lenguaje de manipulación de datos
DMV	dependencia multivaluada
DNF	forma normal disyuntiva
DRDA	Arquitectura de Base de Datos Relacional Distribuida (IBM)
DSL	sublenguaje de datos
DSS	sistema de apoyo para la toma de decisiones
DUW	unidad de trabajo distribuida
EB	hexabyte (1024 PB)
EDB	base de datos extensional
E/S	entrada/salida
EMVD	MVD incrustado
E/R	entidad/vínculo
FD	dependencia funcional
FNBC	forma normal de Boyce/Codd
FNCE	forma normal de clave elemental
FN/DC	forma normal de clave de dominio
GB	gigabyte (1024MB)
GIS	sistema de información geográfica
HOLAP	OLAP híbrido
IDB	base de datos intensional
IDMS	Sistema Integrado de Administración de Base de Datos
IEEE	Instituto de Ingenieros Eléctricos y Electrónicos
IMS	Sistema de Administración de Información
IND	dependencia de inclusión
IS	intento compartido (bloqueo); sistemas de información

ISBL	Lenguaje Básico de Sistemas de Información
ISO	Organización Internacional de Estándares
IT	tecnología de la información
IX	intento exclusivo (bloqueo)
J ACM	*Journal of the A CM* (publicación de la ACM)
' JDBC	Conectividad Java para Base de Datos
K	1024 (a veces 1000)
KB	kilobyte (1024 bytes)
LAN	red de área local
LOB	objeto grande
MB	megabyte (1024 KB)
MLS	seguro a multinivel
MOLAP	OLAP multidimensional
NCITS	Comité Nacional de Estándares sobre la Tecnología de la Información (conocido anteriormente como X3)
NCITS/H2	comité de base de datos NCITS (conocido anteriormente como X3H2)
NF^2	NF al cuadrado = NFNF = no primera forma normal (?)
'ODBC	Conectividad Abierta a Bases de Datos
ODMG	Grupo de Administración de Datos de Objetos
ODS	almacén de datos operacionales
OÍD	ID de objeto
OLAP	procesamiento analítico en línea
OLCP	procesamiento complejo en línea
OLDM	administración de decisiones en línea
OLTP	procesamiento de transacciones en línea
OMG	Grupo de Administración de Objetos
OO	orientado a objetos; orientación a objetos
OODB	base de datos orientada a objetos
OOPL	lenguaje de programación orientado a objetos
OQL	Lenguaje de Consulta de Objetos (parte del ODMG)
OSI	Interconexión de Sistemas Abiertos
OSQL	"SQL Objeto"
PB	petabyte (1024 TB)
PC	computadora personal
PJ/NF	forma normal de proyección-junta
PODS	Principles of Database Systems (conferencia de la ACM)
PRTV	vehículo de pruebas relaciónales Peterlee

PSM	módulos de almacenamiento persistente (parte del estándar SQL)
QBE	consulta por ejemplo
QUEL	lenguaje de consulta
RAID	arreglo redundante de discos baratos
RDA	acceso a datos remotos
RDB	base de datos relacional
RDBMS	DBMS relacional
RID	ID de registro o ID de renglón (almacenado)
ROLAP	OLAP relacional
RM/T	modelo relacional/Tasmania
RM/V1	modelo relacional/Versión 1
RM/V2	modelo relacional/Versión 2
RPC	llamada a procedimiento remoto
RR	lectura repetible (DB2)
RUW	unidad de trabajo remota
RVA	atributo valuado por relación
S	compartido (bloqueo)
SIGMOD	Grupo de Interés Especial sobre la Administración de Datos (grupo de interés especial ACM)
SIX	exclusivo de intento compartido (bloqueo) *vea*
SPARC	ANSiySPARC
'SQL	(originalmente) Lenguaje Estructurado de Consultas; a veces Lenguaje de Consulta Estándar
TB TCB	terabyte (1024 GB)
TCP/IP	Base de Computación Confiable
TID	Protocolo de Control de Transmisión/Protocolo Internet
TODS	ID de tupia (almacenado)
TPC	*Transactions on Database Systems* (publicación de la ACM)
U	Concilio de Procesamiento de Transacciones
UDT	actualización (bloqueo)
UML	tipo definido por el usuario
unk	Lenguaje de Modelado Unificado
UNK	*desconocido* (valor de certeza)
uow	desconocido (nulo)
VLDB	unidad de trabajo
VSAM	base de datos muy grande; Very Large Database (conferencia anual)
	Método de Acceso de Almacenamiento Virtual (IBM)

WAL	bitácora de escritura anticipada
WAN	red de área amplia
WFF	fórmula bien formada
WORM	escribir una vez/leer muchas
WYSIWYG	lo que ves es lo que obtienes
X	exclusivo (bloqueo)
X3	*vea* NCITS
X3H2	*vea* NCITS/H2
INF	primera forma normal
2NF	segunda forma normal
2PC	confirmación en dos fases
2PL	bloqueo en dos fases
2VL	lógica de dos valores
20C	lo mismo que 2PC
20L	lo mismo que 2PL
3GL	lenguaje de tercera generación
3VL	lógica de tres valores
3NF	tercera forma normal
4GL	lenguaje de cuarta generación
4NF	cuarta forma normal
4VL	lógica de cuatro valores
5NF	quinta forma normal (lo mismo que PJ/NF)
e	pertenece a
e	operador de comparación (=, <, etcétera)
0	el conjunto vacío
–>	determina funcionalmente
	multidetermina
=	es equivalente a
=►	implica (conector lógico)
h"	implica (símbolo metalingüístico)
	siempre es el caso que (símbolo metalingüístico)
I	fin (de una prueba, ejemplo, etcétera)

índice

PROGRAMAS EDUCATIVOS, S.A. DE C. V.
CALZ. CHABACANO NO 65
COL ASTURIAS, DELG CUAUHTEMOC. C. P. 06850, MEXICO, D. F.

EMPRESA CERTIFICADA POR EL INSTITUTO MEXICANO DE NORMALIZACIÓN Y CERTIFICACIÓN A C. BAJO LAS NORMAS ISO-9002 1994/N MX-CC-004 1995
CON EL NO DE REGISTRO RSC-048
E ISO-14001 1996/NMX-SAA-001 199B IMNCÍ
CON EL NO DE REGISTRO RSAA-003

```
<expresión de tabla>
   ::=  <expresión de tabla con junta> |
        <expresión de tabla sin junta>
                                  junta>

<expresión de tabla con junta>
   ::=  <referencia de tabla>
              [ NATURAL ] JOIN <referencia de tabla> [
              ON <expresión condicional>
               | USING ( <lista de nombres de columna separados con comas> ) ]
        l <referencia de tabla> CROSS JOIN <referencia de tabla> I ( <expresión
        de tabla con junta> )

■Preferencia de tabla>
   ::=  <nombre de tabla> [ [ AS ] <nombre de variable de alcance>
                           [ ( <lista de nombres de columna separados con comas> ) ) ] |
        ( <expresión de tabla> ) [ AS ] <nombre de variable de alcance»
                           [ ( <lista de nombres de columna separados con comas> ) ]
        | <expresión de tabla con junta>

<expresión de tabla sin junta>
   ;:■  <término de tabla sin junta>
        | <expresión de tabla> UNION [ ALL ] I CORRESPONDING
        [ BY ( <lista de nombres de columna separados con comas> ) ] ] <término de tabla>
        | <expresión de tabla> EXCEPT [ ALL ] [ CORRESPONDING
                           [ BY ( <lista de nombres de columna separados con comas> ) ] ]
                           <término de tabla>

<término de tabla sin junta>
   ::=  <primaria de tabla sin junta>
        | <término de tabla> INTERSECT [ ALL ] [ CORRESPONDING
                           [ BY ( <lista de nombres de columna separados con comas> ) ] ]
                           <primaria de tabla>

<término de tabla>
   ::=  <término de tabla sin junta> |
        <expresión de tabla con junta>

<primaria de tabla>
   ::■  «primaria de tabla sin junta> |
        «expresión de tabla con junta>

<primaria de tabla sin junta> ::=  TABLE
   <nombre de tabla> <constructor de
   tabla> <expresión de seleccionar> (
   <expresión de tabla sin junta> )

<constructor de tabla>
   ::=  VALUES <lista de constructores de fila separados con comas>

<constructor de fila>
   ::=  <expresión escalar>
        I ( <lista de expresiones escalares separadas con comas> )
        I ( <expresión de tabla> )

<expresión de seleccionar>
   ::=  SELECT [ ALL | DISTINCT ] <lista de elementos de seleccionar separados con comas>
            FROM <lista_ de referencias de tabla separadas con comas> [ WHERE <expresión
            condicional> )
                  [ GROUP BY <lista de nombres de columna separados con comas> ] [
                     HAVING <expresión condicional> ]

<elemento de seleccionar>
   ::=  «expresión escalar> [ [ AS ] <nombre de columna> ] |
        [ <nombre de variable de alcance>. ] *
```

La sintaxis de <*expresión*(es) *de tabla*> de SQL

```
<expresión condicional»
    ::=   <término condicional>
        | <expresión condicional> OR <término condicional>

<término condicional>
    ::=   <factor condicional>
        | <término condicional> AND <factor condicional>

^factor condicional>
    ::=   [ NOT ] <primario condicional>

<primario condicional>
    ::=   <condición simple>     ( <expresión condicional*
                                )

■«condición simple>
    ::=   <condición de comparación>
          <condición de pertenencia»
          <condición de semejanza»
          <condición de coincidencia>
          <condición de todos o cualquiera>
          <condición de existencia>
          <condición de unicidad>

<condición de comparación>
    ::=   constructor de fila>
              <operador de comparación> <constructor de fila>

<operador de comparación»
            |   |    |          I ◇

<condición de pertenencia»
    ::=   constructor de fila> [ NOT ] IN ( <expresión de tabla» )
        | <expresión escalar» [ NOT ] IN
              ( <lista_de expresiones escalares separada con comas» )

condición de semejanza»
    ::=   <expresión de cadena de caracteres»
              [ NOT ] LIKE <patrón» [ ESCAPE <escape» ]

condición de coincidencia»
    ::=   constructor de fila»
              MATCH UNIQUE ( <expresión de tabla» )

condición de todos o cualquiera»
    ::=   constructor de fila»
          <operador de comparación» ALL ( <expresión de tabla» ) |
          constructor de fila» cperador de comparación» ANY (
          <expresión de tabla» )

condición de existencia»
    ::=   EXISTS ( expresión de tabla» )

condición de unicidad»       ^
    ::=   UNIQUE ( expresión de tabla» )
```

La sintaxis de <*expresión(es)* condicional(es;» de SQL

' or más de 25 años, el libro *Introducción a los sistemas de bases de datos,* de C.J. Date, ha sido el recurso autorizado para los lectores interesados en conocer y comprender los principios de los sistemas de bases de datos. Esta edición revisada sigue ofreciendo una base sólida de los fundamentos de la tecnología de bases de datos, así como algunas ideas sobre el futuro desarrollo de este campo.

El material está organizado en seis partes principales. La parte I j proporciona una introducción extensa sobre los conceptos de los sistemas de bases de datos en general y sobre los sistemas relaciónales en particular. La parte II consiste en una descripción minuciosa del modelo relacional, el cual es el fundamento teórico ; del campo de las bases de datos en su conjunto. La parte III expone

la teoría y la práctica del diseño de bases de datos. La parte IV se ocupa de la administración de transacciones. La parte V muestra cómo los conceptos relaciónales son relevantes para una diversidad de aspectos de la tecnología de bases de datos (la seguridad, las bases de datos distribuidas, los datos temporales, el apoyo a la toma de decisiones, etcétera). Por último, la parte VI describe el impacto que ha tenido la tecnología de objetos en los sistemas de bases de datos.

Esta séptima edición de *Introducción a los sistemas de bases de datos* contiene material que ha sido revisado para mejorar y ampliar el tratamiento de varios temas, incluyendo:

* Material revisado y ampliado sobre el modelo relacional, en particular en las secciones sobre tipos (dominios), valores de relación contra variables de relación, integridad, predicados y vistas
* Material nuevo sobre atributos evaluados por relación, desnormalización, diseño ortogonal y enfoques alternativos para el modelado semántico (incluyendo las "reglas del negocio")
* Capítulos totalmente nuevos que cubren la herencia, el apoyo a la toma de decisiones y las bases de datos temporales
* Dos nuevos apéndices, uno que resume la sintaxis y la semántica de SQL y otro sobre SQL3

Los lectores de este libro obtendrán un conocimiento sólido de la estructura general, los conceptos y los objetivos de los sistemas de bases de datos, y se familiarizarán con los principios teóricos subyacentes a la construcción de dichos sistemas.

ISBN 968-444-419-2
90000
9 789684 444195

Pearson
Educación

Visítenos en:
www.pearsonedlatino.com