

FUNDAMENTOS DE BASES DE DATOS

QUINTA EDICIÓN

Fundamentos de bases de datos

Quinta edición

ABRAHAM SILBERSCHATZ

Universidad de Yale

HENRY F. KORTH

Universidad de Lehigh

S. SUDARSHAN

Instituto tecnológico indio, Bombay

Traducción

FERNANDO SÁENZ PÉREZ

ANTONIO GARCÍA CORDERO

JESÚS CORREAS FERNÁNDEZ

Universidad Complutense de Madrid

Revisión técnica

LUIS GRAU FERNÁNDEZ

Universidad Nacional de Educación a Distancia



MADRID BOGOTÁ BUENOS AIRES CARACAS GUATEMALA LISBOA
MÉXICO NUEVA YORK PANAMÁ SAN JUAN SANTIAGO SAO PAULO
AUCKLAND HAMBURGO LONDRES MILÁN MONTREAL NUEVA DELHI PARÍS
SAN FRANCISCO SIDNEY SINGAPUR ST. LOUIS TOKIO TORONTO

La información contenida en este libro procede de una obra original publicada por The McGraw-Hill Companies. No obstante, McGraw-Hill/Interamericana de España no garantiza la exactitud o perfección de la información publicada. Tampoco asume ningún tipo de garantía sobre los contenidos y las opiniones vertidas en dichos textos.

Este trabajo se publica con el reconocimiento expreso de que se está proporcionando una información, pero no tratando de prestar ningún tipo de servicio profesional o técnico. Los procedimientos y la información que se presentan en este libro tienen sólo la intención de servir como guía general.

McGraw-Hill ha solicitado los permisos oportunos para la realización y el desarrollo de esta obra.

Fundamentos de bases de datos, quinta edición

No está permitida la reproducción total o parcial de este libro, ni su tratamiento informático, ni la transmisión de ninguna forma o por cualquier medio, ya sea electrónico, mecánico, por fotocopia, por registro u otros métodos, sin el permiso previo y por escrito de los titulares del Copyright.



**McGraw-Hill/Interamericana
de España, S.A.U.**

DERECHOS RESERVADOS © 2006, respecto a la quinta edición en español, por
McGRAW-HILL/INTERAMERICANA DE ESPAÑA, S.A.U.

Edificio Valrealty, 1^a planta
Basauri, 17
28023 Aravaca (Madrid)

<http://www.mcgraw-hill.es>
universidad@mcgraw-hill.com

Traducido de la quinta edición en inglés de
Database System Concepts
ISBN: 0-07-295886-3

Copyright © 2006 por The McGraw-Hill Companies, Inc.

ISBN: 84-481-4644-1
Depósito legal: M. 5.043-2006

Editor: Carmelo Sánchez González
Compuesto por: Fernando Sáenz Pérez
Impreso en Fernández Ciudad, S. L.

IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Contenido

Prefacio xv

Capítulo 1 Introducción

1.1 Aplicaciones de los sistemas de bases de datos	1	1.9 Gestión de transacciones	17
1.2 Propósito de los sistemas de bases de datos	2	1.10 Minería y análisis de datos	18
1.3 Visión de los datos	4	1.11 Arquitectura de las bases de datos	19
1.4 Lenguajes de bases de datos	7	1.12 Usuarios y administradores de bases de datos	21
1.5 Bases de datos relacionales	9	1.13 Historia de los sistemas de bases de datos	22
1.6 Diseño de bases de datos	11	1.14 Resumen	24
1.7 Bases de datos basadas en objetos y semiestructuradas	15	Ejercicios	25
1.8 Almacenamiento de datos y consultas	16	Notas bibliográficas	26

PARTE 1 ■ BASES DE DATOS RELACIONALES

Capítulo 2 El modelo relacional

2.1 La estructura de las bases de datos relacionales	29	2.5 Valores nulos	53
2.2 Operaciones fundamentales del álgebra relacional	36	2.6 Modificación de la base de datos	54
2.3 Otras operaciones del álgebra relacional	44	2.7 Resumen	56
2.4 Operaciones del álgebra relacional extendida	48	Ejercicios	57
		Notas bibliográficas	59

Capítulo 3 SQL

3.1 Introducción	61	3.8 Consultas complejas	80
3.2 Definición de datos	62	3.9 Vistas	81
3.3 Estructura básica de las consultas SQL	65	3.10 Modificación de la base de datos	84
3.4 Operaciones sobre conjuntos	71	3.11 Reunión de relaciones**	90
3.5 Funciones de agregación	73	3.12 Resumen	94
3.6 Valores nulos	75	Ejercicios	95
3.7 Subconsultas anidadas	76	Notas bibliográficas	98

Capítulo 4 SQL avanzado

4.1 Tipos de datos y esquemas	101	4.7 Consultas recursivas**	126
4.2 Restricciones de integridad	105	4.8 Características avanzadas de SQL**	129
4.3 Autorización	111	4.9 Resumen	132
4.4 SQL incorporado	112	Ejercicios	133
4.5 SQL dinámico	114	Notas bibliográficas	135
4.6 Funciones y procedimientos**	121		

Capítulo 5 Otros lenguajes relacionales

5.1 El cálculo relacional de tuplas	137	5.5 Resumen	162
5.2 El cálculo relacional de dominios	141	Ejercicios	163
5.3 Query-by-Example	144	Notas bibliográficas	165
5.4 Datalog	151		

PARTE 2 ■ DISEÑO DE BASES DE DATOS

Capítulo 6 Diseño de bases de datos y el modelo E-R

6.1 Visión general del proceso de diseño	169	6.8 Diseño de una base de datos para un banco	197
6.2 El modelo entidad-relación	171	6.9 Reducción a esquemas relacionales	200
6.3 Restricciones	176	6.10 Otros aspectos del diseño de bases de datos	207
6.4 Diagramas entidad-relación	180	6.11 El lenguaje de modelado unificado UML**	210
6.5 Aspectos del diseño entidad-relación	184	6.12 Resumen	212
6.6 Conjuntos de entidades débiles	189	Ejercicios	213
6.7 Características del modelo E-R extendido	190	Notas bibliográficas	217

Capítulo 7 Diseño de bases de datos relacionales

7.1 Características de los buenos diseños relacionales	219	7.6 Descomposición mediante dependencias multivaloradas	244
7.2 Dominios atómicos y la primera forma normal	223	7.7 Más formas normales	248
7.3 Descomposición mediante dependencias funcionales	224	7.8 Proceso de diseño de las bases de datos	248
7.4 Teoría de las dependencias funcionales	231	7.9 Modelado de datos temporales	251
7.5 Algoritmos de descomposición	239	7.10 Resumen	253
		Ejercicios	254
		Notas bibliográficas	257

Capítulo 8 Diseño y desarrollo de aplicaciones

8.1 Interfaces de usuario y herramientas	259	8.7 Autorización en SQL	278
8.2 Interfaces Web para bases de datos	262	8.8 Seguridad de las aplicaciones	285
8.3 Fundamentos de Web	263	8.9 Resumen	291
8.4 Servlets y JSP	267	Ejercicios	293
8.5 Creación de aplicaciones Web de gran tamaño	271	Notas bibliográficas	297
8.6 Disparadores	273		

PARTE 3 ■ BASES DE DATOS ORIENTADAS A OBJETOS Y XML

Capítulo 9 Bases de datos basadas en objetos

9.1 Visión general	301	9.7 Implementación de las características O-R	315
9.2 Tipos de datos complejos	302	9.8 Lenguajes de programación persistentes	316
9.3 Tipos estructurados y herencia en SQL	303	9.9 Sistemas orientados a objetos y sistemas relacionales orientados a objetos	322
9.4 Herencia de tablas	308	9.10 Resumen	323
9.5 Tipos array y multiconjunto en SQL	309	Ejercicios	324
9.6 Identidad de los objetos y tipos de referencia en SQL	313	Notas bibliográficas	327

Capítulo 10 XML

10.1 Motivación	329	10.6 Almacenamiento de datos XML	350
10.2 Estructura de los datos XML	332	10.7 Aplicaciones XML	354
10.3 Esquema de los documentos XML	335	10.8 Resumen	358
10.4 Consulta y transformación	340	Ejercicios	360
10.5 La interfaz de programación de aplicaciones de XML	349	Notas bibliográficas	362

PARTE 4 ■ ALMACENAMIENTO DE DATOS Y CONSULTAS

Capítulo 11 Almacenamiento y estructura de archivos

11.1 Visión general de los medios físicos de almacenamiento	367	11.6 Organización de los archivos	386
11.2 Discos magnéticos	370	11.7 Organización de los registros en archivos	390
11.3 RAID	375	11.8 Almacenamiento con diccionarios de datos	393
11.4 Almacenamiento terciario	382	11.9 Resumen	395
11.5 Acceso al almacenamiento	383	Ejercicios	396
		Notas bibliográficas	398

Capítulo 12 Indexación y asociación

12.1 Conceptos básicos	401	12.8 Comparación de la indexación ordenada y la asociación	431
12.2 Índices ordenados	402	12.9 Índices de mapas de bits	432
12.3 Archivos de índices de árbol B ⁺	408	12.10 Definición de índices en SQL	435
12.4 Archivos de índices de árbol B	417	12.11 Resumen	436
12.5 Accesos bajo varias claves	418	Ejercicios	438
12.6 Asociación estática	421	Notas bibliográficas	440
12.7 Asociación dinámica	426		

Capítulo 13 Procesamiento de consultas

13.1 Visión general	443	13.6 Otras operaciones	463
13.2 Medidas del coste de una consulta	445	13.7 Evaluación de expresiones	466
13.3 Operación selección	446	13.8 Resumen	470
13.4 Ordenación	450	Ejercicios	472
13.5 Operación reunión	452	Notas bibliográficas	473

Capítulo 14 Optimización de consultas

- 14.1 Visión general 475
- 14.2 Transformación de expresiones relacionales 476
- 14.3 Estimación de las estadísticas de los resultados de las expresiones 482
- 14.4 Elección de los planes de evaluación 487
- 14.5 Vistas materializadas** 494
- 14.6 Resumen 498
- Ejercicios 500
- Notas bibliográficas 502

PARTE 5 ■ GESTIÓN DE TRANSACCIONES

Capítulo 15 Transacciones

- 15.1 Concepto de transacción 507
- 15.2 Estados de una transacción 510
- 15.3 Implementación de la atomicidad y la durabilidad 512
- 15.4 Ejecuciones concurrentes 513
- 15.5 Secuencialidad 516
- 15.6 Recuperabilidad 520
- 15.7 Implementación del aislamiento 522
- 15.8 Comprobación de la secuencialidad 522
- 15.9 Resumen 523
- Ejercicios 525
- Notas bibliográficas 527

Capítulo 16 Control de concurrencia

- 16.1 Protocolos basados en el bloqueo 529
- 16.2 Protocolos basados en marcas temporales 539
- 16.3 Protocolos basados en validación 542
- 16.4 Granularidad múltiple 544
- 16.5 Esquemas multiversión 546
- 16.6 Tratamiento de interbloqueos 548
- 16.7 Operaciones para insertar y borrar 553
- 16.8 Niveles débiles de consistencia 555
- 16.9 Concurrencia en los índices** 557
- 16.10 Resumen 560
- Ejercicios 562
- Notas bibliográficas 566

Capítulo 17 Sistema de recuperación

- 17.1 Clasificación de los fallos 567
- 17.2 Estructura del almacenamiento 568
- 17.3 Recuperación y atomicidad 571
- 17.4 Recuperación basada en el registro histórico 572
- 17.5 Transacciones concurrentes y recuperación 579
- 17.6 Gestión de la memoria intermedia 581
- 17.7 Fallo con pérdida de almacenamiento no volátil 584
- 17.8 Técnicas avanzadas de recuperación** 585
- 17.9 Sistemas remotos de copias de seguridad 591
- 17.10 Resumen 593
- Ejercicios 596
- Notas bibliográficas 597

PARTE 6 ■ MINERÍA DE DATOS Y RECUPERACIÓN DE INFORMACIÓN

Capítulo 18 Análisis y minería de datos

- 18.1 Sistemas de ayuda a la toma de decisiones 601
- 18.2 Análisis de datos y OLAP 602
- 18.3 Almacenes de datos 612
- 18.4 Minería de datos 615
- 18.5 Resumen 625
- Ejercicios 627
- Notas bibliográficas 628

Capítulo 19 Recuperación de información

19.1 Visión general 631	19.7 Motores de búsqueda en Web 641
19.2 Clasificación por relevancia según los términos 632	19.8 Recuperación de información y datos estructurados 642
19.3 Relevancia según los hipervínculos 635	19.9 Directorios 643
19.4 Sinónimos, homónimos y ontologías 638	19.10 Resumen 645
19.5 Creación de índices de documentos 639	Ejercicios 646
19.6 Medida de la efectividad de la recuperación 640	Notas bibliográficas 647

PARTE 7 ■ ARQUITECTURA DE SISTEMAS

Capítulo 20 Arquitecturas de los sistemas de bases de datos

20.1 Arquitecturas centralizadas y cliente-servidor 651	20.5 Tipos de redes 666
20.2 Arquitecturas de sistemas servidores 653	20.6 Resumen 668
20.3 Sistemas paralelos 657	Ejercicios 669
20.4 Sistemas distribuidos 663	Notas bibliográficas 671

Capítulo 21 Bases de datos paralelas

21.1 Introducción 673	21.6 Paralelismo entre operaciones 685
21.2 Paralelismo de E/S 674	21.7 Diseño de sistemas paralelos 687
21.3 Paralelismo entre consultas 677	21.8 Resumen 688
21.4 Paralelismo en consultas 678	Ejercicios 689
21.5 Paralelismo en operaciones 678	Notas bibliográficas 691

Capítulo 22 Bases de datos distribuidas

22.1 Bases de datos homogéneas y heterogéneas 693	22.7 Procesamiento distribuido de consultas 714
22.2 Almacenamiento distribuido de datos 694	22.8 Bases de datos distribuidas heterogéneas 717
22.3 Transacciones distribuidas 697	22.9 Sistemas de directorio 719
22.4 Protocolos de compromiso 698	22.10 Resumen 723
22.5 Control de la concurrencia en las bases de datos distribuidas 704	Ejercicios 726
22.6 Disponibilidad 710	Notas bibliográficas 728

PARTE 8 ■ OTROS TEMAS

Capítulo 23 Desarrollo avanzado de aplicaciones

23.1 Ajuste del rendimiento 733	23.5 Resumen 749
23.2 Pruebas de rendimiento 741	Ejercicios 750
23.3 Normalización 744	Notas bibliográficas 751
23.4 Migración de aplicaciones 748	

Capítulo 24 Tipos de datos avanzados y nuevas aplicaciones

24.1 Motivación	753	24.5 Computadoras portátiles y bases de datos personales	767
24.2 El tiempo en las bases de datos	754	24.6 Resumen	772
24.3 Datos espaciales y geográficos	756	Ejercicios	773
24.4 Bases de datos multimedia	765	Notas bibliográficas	775

Capítulo 25 Procesamiento avanzado de transacciones

25.1 Monitores de procesamiento de transacciones	777	25.6 Transacciones de larga duración	791
25.2 Flujos de trabajo de transacciones	781	25.7 Gestión de transacciones en varias bases de datos	796
25.3 Comercio electrónico	786	25.8 Resumen	799
25.4 Bases de datos en memoria principal	788	Ejercicios	801
25.5 Sistemas de transacciones de tiempo real	790	Notas bibliográficas	802

PARTE 9 ■ ESTUDIO DE CASOS

Capítulo 26 PostgreSQL

Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou, Bianca Schroeder

26.1 Introducción	807	26.5 Almacenamiento e índices	824
26.2 Interfaces de usuario	808	26.6 Procesamiento y optimización de consultas	827
26.3 Variaciones y extensiones de SQL	809	26.7 Arquitectura del sistema	830
26.4 Gestión de transacciones en PostgreSQL	817	Notas bibliográficas	831

Capítulo 27 Oracle

Hakan Jakobsson

27.1 Herramientas para el diseño de bases de datos y la consulta	833	27.6 Arquitectura del sistema	851
27.2 Variaciones y extensiones de SQL	834	27.7 Réplica, distribución y datos externos	854
27.3 Almacenamiento e índices	836	27.8 Herramientas de gestión de bases de datos	855
27.4 Procesamiento y optimización de consultas	844	27.9 Minería de datos	856
27.5 Control de concurrencia y recuperación	849	Notas bibliográficas	857

Capítulo 28 DB2 Universal Database de IBM

Sriram Padmanabhan

28.1 Visión general	859	28.8 Características autónomas de DB2	876
28.2 Herramientas de diseño de bases de datos	860	28.9 Herramientas y utilidades	876
28.3 Variaciones y extensiones de SQL	861	28.10 Control de concurrencia y recuperación	878
28.4 Almacenamiento e indexación	864	28.11 Arquitectura del sistema	880
28.5 Agrupación multidimensional	867	28.12 Réplicas, distribución y datos externos	881
28.6 Procesamiento y optimización de consultas	870	28.13 Características de inteligencia de negocio	882
28.7 Tablas de consultas materializadas	874	Notas bibliográficas	882

Capítulo 29 SQL Server de Microsoft

Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas, Michael Zwilling

29.1 Herramientas para la administración, el diseño y la consulta de las bases de datos	885	29.8 Procesamiento de consultas heterogéneas distribuidas	905
29.2 Variaciones y extensiones de SQL	889	29.9 Duplicación	906
29.3 Almacenamiento e índices	892	29.10 Programación de servidores en .NET	908
29.4 Procesamiento y optimización de consultas	895	29.11 Soporte de XML en SQL Server 2005	912
29.5 Conurrencia y recuperación	899	29.12 Service Broker de SQLServer	916
29.6 Arquitectura del sistema	903	29.13 Almacenes de datos e inteligencia de negocio	918
29.7 Acceso a los datos	904	Notas bibliográficas	921

Bibliografía 923

Índice 943

Prefacio

La gestión de las bases de datos ha evolucionado desde una aplicación informática especializada hasta convertirse en parte esencial de los entornos informáticos modernos. Por tanto, el conocimiento acerca de los sistemas de bases de datos se ha convertido en una parte imprescindible de la formación en informática. En este texto se presentan los conceptos fundamentales de la gestión de las bases de datos. Estos conceptos incluyen aspectos del diseño de bases de datos, de los lenguajes y de la implementación de los sistemas de bases de datos.

Este libro está orientado a un primer curso de bases de datos para los niveles técnicos y superiores. Además del material básico para un primer curso, el texto también contiene material avanzado que se puede usar como complemento del curso o como material introductorio de un curso avanzado.

En este libro se asume que se dispone de conocimientos básicos sobre estructuras de datos básicas, organización de computadoras y un lenguaje de programación de alto nivel (tipo Pascal). Los conceptos se presentan usando descripciones intuitivas, muchas de las cuales están basadas en el ejemplo propuesto de una entidad bancaria. Se tratan los resultados teóricos importantes, pero se omiten las demostraciones formales. En lugar de las demostraciones se usan figuras y ejemplos para sugerir su justificación. Las descripciones formales y las pruebas pueden hallarse en los artículos de investigación y en los textos avanzados a los que se hace referencia en las notas bibliográficas.

Los conceptos y algoritmos fundamentales tratados en este libro suelen basarse en los usados en sistemas de bases de datos comerciales o experimentales ya existentes. El objetivo es presentar esos conceptos y algoritmos en un marco general que no esté vinculado a ningún sistema de bases de datos en concreto. Los detalles de los sistemas de bases de datos concretos se estudian en la Parte 9, “Estudio de casos”.

En esta quinta edición de *Fundamentos de bases de datos* se ha mantenido el estilo global de las ediciones anteriores, mientras que su contenido y organización han evolucionado para reflejar las modificaciones que se están produciendo en el modo en que las bases de datos se diseñan, se gestionan y se usan. También se han tenido en cuenta las tendencias en la enseñanza de los conceptos de bases de datos y se han hecho adaptaciones para facilitar esas tendencias donde ha resultado conveniente. Antes de describir el contenido del libro con detalle, se destacarán algunas de las características de la quinta edición.

- **Tratamiento precoz de SQL.** Muchos profesores usan SQL como componente principal de sus proyectos de fin de curso (visítese el sitio Web <http://www.mhe.es/universidad/informatica/fundamentos> para ver proyectos de ejemplo). Con objeto de proporcionar a los estudiantes tiempo suficiente para realizar los proyectos, especialmente en universidades que trabajen por trimestres, resulta fundamental enseñar SQL tan pronto como sea posible. Con esto en mente se han llevado a cabo varias modificaciones en la organización del libro:
 1. Aplazar la presentación del modelo entidad-relación hasta la Parte 2, titulada “Diseño de bases de datos”.

2. Racionalizar la introducción del modelo relacional mediante el aplazamiento del tratamiento del cálculo relacional hasta el Capítulo 5, mientras se mantiene el tratamiento del álgebra relacional en el Capítulo 2.
3. Dedicar dos de los primeros capítulos a SQL. El Capítulo 3 trata las características básicas de SQL, incluidas la definición y la manipulación de los datos. El Capítulo 4 trata características más avanzadas, como las restricciones de integridad, SQL dinámico y los constructores procedimentales. Entre el material nuevo de este capítulo figura un tratamiento más amplio de JDBC, de los constructores procedimentales en SQL, de la recursión en SQL y de las nuevas características de SQL:2003. Este capítulo también incluye una breve visión general de las autorizaciones, aunque su tratamiento detallado se pospone hasta el Capítulo 8.

Estas modificaciones permiten que los estudiantes comiencen a escribir consultas de SQL en una etapa inicial del curso y que se familiaricen con el empleo de los sistemas de bases de datos. Esto también permite a los estudiantes desarrollar una intuición sobre el diseño de bases de datos que facilita la enseñanza de las metodologías de diseño en la Parte 2 del texto. Se ha descubierto que los estudiantes aprecian mejor los aspectos del diseño de bases de datos con esta organización.

- **Una parte nueva (la Parte 2) que está dedicada al diseño de las bases de datos.** La Parte 2 del texto contiene tres capítulos dedicados al diseño de las bases de datos y de las aplicaciones para bases de datos. Aquí se incluye un capítulo (el Capítulo 6) sobre el modelo entidad-relación que contiene todo el material del capítulo correspondiente de la cuarta edición (el Capítulo 2), además de varias actualizaciones significativas. También se presenta en el Capítulo 6 una breve visión general del proceso de diseño de las bases de datos. Los profesores que prefieran comenzar el curso con el modelo E-R pueden empezar por este capítulo sin pérdida de continuidad, ya que se ha hecho todo lo posible para evitar las dependencias con cualquier capítulo anterior salvo con el Capítulo 1.

El Capítulo 7, sobre el diseño relacional, presenta el material tratado en el Capítulo 7 de la cuarta edición, pero con un estilo nuevo y más legible. Los conceptos de diseño del modelo E-R se usan para crear una visión general intuitiva de los aspectos del diseño relacional, adelantándose a la presentación del enfoque formal al diseño mediante dependencias funcionales y multivaloradas y la normalización algorítmica. Este capítulo también incluye un apartado nuevo sobre los aspectos temporales del diseño de bases de datos.

La Parte 2 concluye con un nuevo capítulo, el Capítulo 8, que describe el diseño y el desarrollo de las aplicaciones de bases de datos, incluidas las aplicaciones Web, los servlets, JSP, los disparadores y los aspectos de seguridad. Para atender a la creciente necesidad de proteger el software de ataques, el tratamiento de la seguridad ha aumentado significativamente respecto de la cuarta edición.

- **Tratamiento completamente revisado y actualizado de las bases de datos basadas en objetos y de XML.** La Parte 3 incluye un capítulo profundamente revisado sobre las bases de datos basadas en objetos que pone el énfasis en las características relacionales de objetos de SQL y sustituye a los capítulos independientes sobre bases de datos orientadas a objetos y bases de datos relacionales de objetos de la cuarta edición. Se ha eliminado parte del material introductorio sobre la orientación a objetos de cursos anteriores con el que los estudiantes están familiarizados, al igual que los detalles sintácticos del ahora obsoleto estándar ODMG. No obstante, se han conservado conceptos importantes subyacentes a las bases de datos orientadas a objetos, incluyendo nuevo material sobre el estándar JDO para añadir persistencia a Java.

La Parte 3 incluye también un capítulo sobre el diseño y la consulta de datos XML, que se ha revisado a fondo respecto del capítulo correspondiente de la cuarta edición. Contiene un tratamiento mejorado de XML Schema y de XQuery, tratamiento del estándar SQL/XML y más ejemplos de aplicaciones XML, incluyendo servicios Web.

- **Material reorganizado sobre la minería de datos y la recuperación de la información.** La minería de datos y el procesamiento analítico en conexión son actualmente usos extremadamente importantes de las bases de datos—ya no sólo “temas avanzados”. Por tanto, se ha trasladado el

tratamiento de estos temas a una parte nueva, la Parte 6, que contiene un capítulo sobre minería y análisis de datos junto con otro capítulo sobre la recuperación de la información.

- **Nuevo caso de estudio: PostgreSQL.** PostgreSQL es un sistema de bases de datos de código abierto que ha conseguido enorme popularidad en los últimos años. Además de ser una plataforma sobre la que crear aplicaciones de bases de datos, el código fuente puede estudiarse y ampliarse en cursos que pongan énfasis en la implementación de sistemas de bases de datos. Por tanto, se ha añadido un caso de estudio de PostgreSQL a la Parte 9, en la que se suma a los tres casos de estudio que aparecían en la cuarta edición (Oracle, IBM DB2 y Microsoft SQL Server). Estos tres últimos casos de estudio se han actualizado para que reflejen las últimas versiones del software respectivo.

El tratamiento de los temas que no se han mencionado anteriormente, incluidos el procesamiento de transacciones (conurrencia y recuperación), las estructuras de almacenamiento, el procesamiento de consultas y las bases de datos distribuidas y paralelas, se ha actualizado respecto de sus equivalentes de la cuarta edición, aunque su organización general permanezca relativamente intacta. El tratamiento de QBE en el Capítulo 5 se ha revisado, eliminando los detalles sintácticos de la agregación y de las actualizaciones que no corresponden a ninguna implementación real, pero se han conservado los conceptos fundamentales de QBE.

Organización

El texto está organizado en nueve partes principales y tres apéndices.

- **Visión general** (Capítulo 1). En el Capítulo 1 se proporciona una visión general de la naturaleza y propósito de los sistemas de bases de datos. Se explica cómo se ha desarrollado el concepto de sistema de bases de datos, cuáles son las características usuales de los sistemas de bases de datos, lo que proporciona al usuario un sistema de bases de datos y cómo se comunican los sistemas de bases de datos con los sistemas operativos. También se introduce un ejemplo de aplicación de las bases de datos: una entidad bancaria que consta de muchas sucursales. Este ejemplo se usa a lo largo de todo el libro. Este capítulo es de naturaleza justificativa, histórica y explicativa.
- **Parte 1: Bases de datos relacionales** (Capítulos 2 a 5). El Capítulo 2 introduce el modelo relacional de datos y trata de conceptos básicos y del álgebra relacional. Este capítulo también ofrece una breve introducción a las restricciones de integridad. Los Capítulos 3 y 4 se centran en el más influyente de los lenguajes relacionales orientados al usuario: SQL. Mientras que el Capítulo 3 ofrece una introducción básica a SQL, el Capítulo 4 describe características de SQL más avanzadas, incluido el modo de establecer comunicación entre un lenguaje de programación y una base de datos que soporte SQL. El Capítulo 5 trata de otros lenguajes relacionales, incluido el cálculo relacional, QBE y Datalog.

Los capítulos de esta parte describen la manipulación de los datos: consultas, actualizaciones, inserciones y eliminaciones y dan por supuesto que se ha proporcionado un diseño de esquema. Los aspectos del diseño de esquemas se posponen hasta la Parte 2.

- **Parte 2: Diseño de bases de datos** (Capítulos 6 a 8). El Capítulo 6 ofrece una visión general del proceso de diseño de las bases de datos, con el énfasis puesto en el diseño mediante el modelo de datos entidad-relación. Este modelo ofrece una vista de alto nivel de los aspectos del diseño de las bases de datos y de los problemas que se hallan al capturar la semántica de las aplicaciones realistas en las restricciones de un modelo de datos. La notación de los diagramas de clase UML también se trata en este capítulo.

El Capítulo 7 introduce la teoría del diseño de las bases de datos relacionales. Se tratan la teoría de las dependencias funcionales y de la normalización, con el énfasis puesto en la motivación y la comprensión intuitiva de cada forma normal. Este capítulo comienza con una visión general del diseño relacional y se basa en la comprensión intuitiva de la implicación lógica de las dependencias funcionales. Esto permite introducir el concepto de normalización antes de haber tratado completamente la teoría de la dependencia funcional, que se presenta más avanzado el capítulo.

Los profesores pueden decidir usar únicamente este tratamiento inicial de los Apartados 7.1 a 7.3 sin pérdida de continuidad. Los profesores que empleen todo el capítulo conseguirán que los estudiantes tengan una buena comprensión de los conceptos de normalización para justificar algunos de los conceptos más difíciles de comprender de la teoría de la dependencia funcional.

El Capítulo 8 trata del diseño y del desarrollo de las aplicaciones. Este capítulo pone énfasis en la creación de aplicaciones de bases de datos con interfaces basadas en Web. Además, el capítulo trata de la seguridad de las aplicaciones.

- **Parte 3: Bases de datos basadas en objetos y XML** (Capítulos 9 y 10). El Capítulo 9 trata de las bases de datos basadas en objetos. Este capítulo describe el modelo de datos objeto-relación, que amplía el modelo de datos relacional para dar soporte a tipos de datos complejos, la herencia de tipos y la identidad de los objetos. Este capítulo también describe el acceso a las bases de datos desde lenguajes de programación orientados a objetos.

El Capítulo 10 trata del estándar XML para la representación de datos, que está experimentando un uso creciente en el intercambio y el almacenamiento de datos complejos. Este capítulo también describe los lenguajes de consultas para XML.

- **Parte 4: Almacenamiento de datos y consultas** (Capítulos 11 a 14). El Capítulo 11 trata de la estructura de disco, de archivos y del sistema de archivos. En el Capítulo 12 se presenta una gran variedad de técnicas de acceso a los datos, incluidos los índices asociativos y de árbol B⁺. Los Capítulos 13 y 14 abordan los algoritmos de evaluación de consultas y su optimización. En estos capítulos se examinan los aspectos internos de los componentes de almacenamiento y de recuperación de las bases de datos.

- **Parte 5: Gestión de transacciones** (Capítulos 15 a 17). El Capítulo 15 se centra en los fundamentos de los sistemas de procesamiento de transacciones, incluidas la atomicidad, la consistencia, el aislamiento y la durabilidad de las transacciones, así como la noción de la secuencialidad.

El Capítulo 16 se centra en el control de concurrencia y presenta varias técnicas para garantizar la secuencialidad, incluidos el bloqueo, las marcas de tiempo y las técnicas optimistas (de validación). Este capítulo también trata los interbloqueos.

El Capítulo 17 trata las principales técnicas para garantizar la ejecución correcta de las transacciones pese a las caídas del sistema y los fallos de los discos. Estas técnicas incluyen los registros, los puntos de revisión y los volcados de las bases de datos.

- **Parte 6: Minería de datos y recuperación de la información** (Capítulos 18 y 19). El Capítulo 18 introduce el concepto de almacén de datos y explica la minería de datos y el procesamiento analítico en conexión (*online analytical processing*, OLAP), incluido el soporte de OLAP y del almacenamiento de datos por SQL:1999. El Capítulo 19 describe las técnicas de recuperación de datos para la consulta de datos textuales, incluidas las técnicas basadas en hipervínculos usadas en los motores de búsqueda Web.

La Parte 6 usa los conceptos de modelado y de lenguaje de las partes 1 y 2, pero no depende de las partes 3, 4 o 5. Por tanto, puede incorporarse fácilmente en cursos que se centren en SQL y en el diseño de bases de datos.

- **Parte 7: Arquitectura de sistemas** (Capítulos 20 a 22). El Capítulo 20 trata de la arquitectura de los sistemas informáticos y describe la influencia de los subyacentes a los sistemas de bases de datos. En este capítulo se estudian los sistemas centralizados, los sistemas cliente–servidor, las arquitecturas paralela y distribuida, y los tipos de red.

El Capítulo 21, que trata las bases de datos paralelas, explora una gran variedad de técnicas de paralelismo, incluidos el paralelismo de E/S, el paralelismo en consultas y entre consultas, y el paralelismo en operaciones y entre operaciones. Este capítulo también describe el diseño de sistemas paralelos.

El Capítulo 22 trata de los sistemas distribuidos de bases de datos, revisitando los aspectos del diseño de bases de datos, la gestión de las transacciones y la evaluación y la optimización de las consultas en el contexto de las bases de datos distribuidas. Este capítulo también trata aspectos de la disponibilidad de los sistemas durante los fallos y describe el sistema de directorios LDAP.

- **Parte 8: Otros temas** (Capítulos 23 a 25). El Capítulo 23 trata de las pruebas de rendimiento, el ajuste de rendimiento, la normalización y la migración de aplicaciones desde sistemas heredados.

El Capítulo 24 trata de los tipos de datos avanzados y de las nuevas aplicaciones, incluidos los datos temporales, los datos espaciales y geográficos, y los aspectos de la gestión de bases de datos móviles y personales.

Finalmente, el Capítulo 25 trata del procesamiento avanzado de transacciones. Entre los temas tratados están los monitores de procesamiento de transacciones, los flujos de trabajo transaccionales, el comercio electrónico, los sistemas de transacciones de alto rendimiento, los sistemas de transacciones en tiempo real, las transacciones de larga duración y la gestión de transacciones en sistemas con múltiples bases de datos.

- **Parte 9: Estudio de casos** (Capítulos 26 a 29). En esta parte se estudian cuatro de los principales sistemas de bases de datos, como PostgreSQL, Oracle, DB2 de IBM y SQL Server de Microsoft. Estos capítulos destacan las características propias de cada uno de los sistemas y describen su estructura interna. Ofrecen gran abundancia de información interesante sobre los productos respectivos y ayudan al lector a comprender el uso en los sistemas reales de las diferentes técnicas de implementación descritas en partes anteriores. También tratan varios aspectos prácticos interesantes del diseño de sistemas reales.

- **Apéndices en Internet.** Aunque la mayor parte de las nuevas aplicaciones de bases de datos usan el modelo relacional o el modelo relacional orientado a objetos, los modelos de datos de red y jerárquico se siguen usando en algunas aplicaciones heredadas. Para los lectores interesados en estos modelos de datos se ofrecen apéndices que describen los modelos de datos de red y jerárquico, en los Apéndices A y B respectivamente; los apéndices sólo se encuentran disponibles en la dirección <http://www.mhe.es/universidad/informatica/fundamentos>. Estos apéndices sólo están disponibles en inglés.

El Apéndice C describe el diseño avanzado de bases de datos relacionales, incluida la teoría de las dependencias multivaloradas, las dependencias de reunión y las formas normales de proyección-reunión y de dominio-clave. Este apéndice está pensado para quienes deseen estudiar la teoría del diseño de bases de datos relacionales con más detalle y para los profesores que deseen hacerlo en sus cursos. De nuevo, este apéndice está disponible únicamente en la página Web del libro.

La quinta edición

La producción de esta quinta edición ha estado guiada por los muchos comentarios y sugerencias que se han recibido relativos a ediciones anteriores, por nuestras propias observaciones al ejercer la docencia en la Universidad de Yale, la Universidad de Lehigh y el IIT de Bombay y por nuestro análisis de las direcciones hacia las que está evolucionando la tecnología de las bases de datos.

El procedimiento básico fue reescribir el material de cada capítulo, actualizando el material antiguo, añadiendo explicaciones sobre desarrollos recientes de la tecnología de las bases de datos y mejorando las descripciones de los temas que los estudiantes hallaban difíciles de comprender. Al igual que en la cuarta edición, cada capítulo tiene una lista de términos de repaso que puede ayudar a los lectores a repasar los temas principales que se han tratado en él. La mayor parte de los capítulos también contiene al final una sección que ofrece información sobre las herramientas de software relacionadas con el tema del capítulo. También se han añadido ejercicios nuevos y referencias actualizadas.

Nota para los profesores

Este libro contiene material tanto básico como avanzado, que puede que no se abarque en un solo semestre. Se han marcado varios apartados como avanzados mediante el símbolo **. Estos apartados pueden omitirse, si se desea, sin pérdida de continuidad. Los ejercicios que son difíciles (y pueden omitirse) también están marcados mediante el símbolo ***.

Es posible diseñar los cursos usando varios subconjuntos de los capítulos. A continuación se esbozan varias de las posibilidades:

- Los apartados del Capítulo 4 desde el Apartado 4.6 en adelante pueden omitirse en los cursos introductorios.
- El Capítulo 5 puede omitirse si los estudiantes no van a usar el cálculo relacional, QBE ni Datalog como parte del curso.
- Los Capítulos 9 (Bases de datos orientadas a objetos), 10 (XML) y 14 (Optimización de consultas) pueden omitirse en los cursos introductorios.
- Tanto el tratamiento del procesamiento de las transacciones (Capítulos 15 a 17) como el tratamiento de la arquitectura de los sistemas de bases de datos (Capítulos 20 a 22) constan de un capítulo introductorio (Capítulos 15 y 20, respectivamente) seguidos de los capítulos con los detalles. Se puede decidir usar los Capítulos 15 y 20 y omitir los Capítulos 16, 17, 21 y 22, si se posponen esos capítulos para un curso avanzado.
- Los Capítulos 18 y 19, que tratan de la minería de datos y de la recuperación de la información, pueden usarse como material para estudio individual u omitirse en los cursos introductorios.
- Los Capítulos 23 a 25 son adecuados para cursos avanzados o para su estudio individual por parte de los estudiantes.
- Los capítulos de estudio de casos 26 a 29 son adecuados para su estudio individual por los estudiantes.

Se pueden encontrar modelos de programaciones para cursos, basados en este texto, en la página Web del libro:

<http://www.mhe.es/universidad/informatica/fundamentos>

Cómo entrar en contacto con los autores y otros usuarios

Se ha hecho todo lo posible por eliminar del texto las erratas y errores. Pero, como en los nuevos desarrollos de software, probablemente queden fallos. En la página Web del libro se puede obtener una lista de erratas actualizada. Se agradecerá que se notifique cualquier error u omisión del libro que no se encuentre en la lista de erratas actual.

Nos alegrará recibir sugerencias de mejoras para el libro. También son bienvenidas las contribuciones para la página Web del libro que puedan resultar útiles para otros lectores, como ejercicios de programación, sugerencias de proyectos, laboratorios y tutoriales en línea, y ayudas a la docencia.

El correo electrónico debe dirigirse a db-book@cs.yale.edu. El resto de la correspondencia debe enviarse a Avi Silberschatz, Department of Computer Science, Yale University, 51 Prospect Street, P.O. Box 208285, New Haven, CT 06520-8285, EE.UU.

También se dispone de una lista de correo mediante la cual los usuarios del libro pueden comunicarse entre sí y con los autores, y recibir actualizaciones del libro y otra información relacionada. La lista está moderada, por lo que en ella no se recibe correo basura. Se ruega seguir el enlace a la lista de correo de la página Web del libro para suscribirse.

Agradecimientos

Esta edición se ha beneficiado de los muchos y útiles comentarios que nos han proporcionado los estudiantes que han usado las cuatro ediciones anteriores. Además, muchas personas nos han escrito o hablado acerca del libro, y nos han ofrecido sugerencias y comentarios. Aunque no podemos mencionarlos aquí a todos, damos las gracias especialmente a los siguientes:

- A Hani Abu-Salem, Universidad DePaul; Jamel R. Alsabagh, Universidad Grand Valley State; Ramzi Bualuan, Universidad Notre Dame; Zhengxin Chen, Universidad de Nebraska en Omaha; Jan Chomick, Universidad SUNY Buffalo; Qin Ding, Universidad Penn State en Harrisburg; Frantisek Franek, Universidad McMaster; Shashi K. Gadia, Universidad Iowa State; William Hankley, Universidad Kansas State; Randy M. Kaplan, Universidad Drexel; Mark Llewellyn, Universidad

de Central Florida; Marty Maskarinec, Universidad Western Illinois; Yiu-Kai Dennis Ng, Universidad Brigham Young; Sunil Prabhakar, Universidad Purdue; Stewart Shen, Universidad Old Dominion; Anita Whitehall, Foothill College; Christopher Wilson, Universidad de Oregón; Weining Zhang, Universidad de Texas en San Antonio; todos ellos revisaron el libro y sus comentarios nos ayudaron mucho a formular esta quinta edición.

- A Anastassia Ailamaki, Sailesh Krishnamurthy, Spiros Papadimitriou y Bianca Schroeder (Universidad Carnegie Mellon) por escribir el apéndice que describe el sistema de bases de datos PostgreSQL.
- A Hakan Jakobsson (Oracle) por el apéndice sobre el sistema de bases de datos Oracle.
- A Sriram Padmanabhan (IBM) por el apéndice que describe el sistema de bases de datos DB2 de IBM.
- A Sameet Agarwal, José A. Blakeley, Thierry D'Hers, Gerald Hinson, Dirk Myers, Vaqar Pirzada, Bill Ramos, Balaji Rathakrishnan, Michael Rys, Florian Waas y Michael Zwilling (todos ellos de Microsoft) por el apéndice sobre el sistema de bases de datos Microsoft SQL Server. A José Blakeley también por coordinar y editar el Capítulo 29, y a César Galindo-Legaria, Goetz Graefe, Kalen Delaney y Thomas Casey (todos ellos de Microsoft) por sus aportaciones a la edición anterior del capítulo sobre SQL Server de Microsoft.
- A Chen Li y a Sharad Mehrotra por ofrecer material sobre JDBC y sobre seguridad que nos ha ayudado a actualizar y ampliar el Capítulo 8.
- A Valentin Dinu, Goetz Graefe, Bruce Hillyer, Chad Hogg, Nahid Rahman, Patrick Schmid, Jeff Storey, Prem Thomas, Liu Zhenming y, especialmente, a N.L. Sarda por su respuesta, que nos ayudó a preparar la quinta edición.
- A Rami Khouri, Nahid Rahman y Michael Rys por su respuesta a las versiones de borrador de los capítulos de la quinta edición.
- A Raj Ashar, Janek Bogucki, Gavin M. Bierman, Christian Breimann, Tom Chappell, Y. C. Chin, Laurens Damen, Prasanna Dhandapani, Arvind Hulgeri, Zheng Jiaping, Graham J. L. Kemp, Hae Choon Lee, Sang-Won Lee, Thanh-Duy Nguyen, D. B. Phatak, Juan Altmayer Pizzorno, Rajarshi Rakshit, Greg Riccardi, N. L. Sarda, Max Smolens, Nikhil Sethi y Tim Wahls por señalar errores en la cuarta edición.
- A Marilyn Turnamian, cuya excelente ayuda como secretaria fue esencial para la finalización a tiempo de esta quinta edición.

La editora fue Betsy Jones. La editora patrocinadora fue Kelly Lowery. La editora de desarrollo fue Melinda D. Bilecki. La directora del proyecto fue Peggy Selle. El director ejecutivo de mercadotecnia fue Michael Weitz. La directora de mercadotecnia fue Dawn Bercier. La ilustradora y diseñadora de la cubierta fue JoAnne Schopler. El editor de copia autónomo fue George Watson. La correctora de pruebas autónoma fue Judy Ganterbein. La diseñadora fue Laurie Janssen. El indexador autónomo fue Tobiah Waldron.

Esta edición se basa en las cuatro ediciones anteriores, por lo que volvemos a dar las gracias a las muchas personas que nos ayudaron con las cuatro primeras ediciones, como R. B. Abhyankar, Don Battory, Phil Bernhard, Haran Boral, Paul Bourgeois, Phil Bohannon, Robert Brazile, Yuri Breitbart, Michael Carey, Soumen Chakrabarti, J. Edwards, Christos Faloutsos, Homma Farian, Alan Fekete, Shashi Gaadia, Jim Gray, Le Gruenwald, Eitan M. Gurari, Ron Hitchens, Yannis Ioannidis, Hyoung-Joo Kim, Won Kim, Henry Korth (padre de Henry F.), Carol Kroll, Gary Lindstrom, Irwin Levinstein, Ling Liu, Dave Maier, Keith Marzullo, Fletcher Mattox, Sharad Mehrotra, Jim Melton, Alberto Mendelzon, Hector Garcia-Molina, Ami Motro, Bhagirath Narahari, Anil Nigam, Cyril Orji, Meral Ozsoyoglu, Bruce Porter, Jim Peterson, K.V. Raghavan, Krithi Ramamritham, Mike Reiter, Odinaldo Rodriguez, Mark Roth, Marek Rusinkiewicz, Sunita Sarawagi, N.L. Sarda, S. Seshadri, Shashi Shekhar, Amit Sheth, Nandit Soparkar, Greg Speegle, Dilys Thomas y Marianne Winslett.

Marilyn Turnamian y Nandprasad Joshi ofrecieron ayuda como secretarias para la cuarta edición, y Marilyn también preparó un primer borrador del diseño de la cubierta para la cuarta edición. Lyn Dupré editó la copia de la tercera edición y Sara Strandtmann editó el texto de la tercera edición. Nilesh Dalvi, Sumit Sanghai, Gaurav Bhalotia, Arvind Hulgeri K.V. Raghavan, Prateek Kapadia, Sara Strandtmann, Greg Speegle y Dawn Bezviner ayudaron a preparar el manual para los profesores de ediciones anteriores. La nueva cubierta es una evolución de las cuatro primeras ediciones. La idea de usar barcos como parte del concepto de la cubierta nos la sugirió inicialmente Bruce Stephan.

Finalmente, Sudarshan quiere agradecer a su esposa, Sita, su amor y su apoyo, y a su hijo Madhur su amor. Hank quiere agradecer a su mujer, Joan, y a sus hijos, Abby y Joe, su amor y su comprensión. Avi quiere agradecer a Valerie su amor, su paciencia y su apoyo durante la revisión de este libro.

A. S.

H. F. K.

S. S.

Introducción

Un **sistema gestor de bases de datos** (SGBD) consiste en una colección de datos interrelacionados y un conjunto de programas para acceder a dichos datos. La colección de datos, normalmente denominada **base de datos**, contiene información relevante para una empresa. El objetivo principal de un SGBD es proporcionar una forma de almacenar y recuperar la información de una base de datos de manera que sea tanto *práctica* como *eficiente*.

Los sistemas de bases de datos se diseñan para gestionar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para almacenar la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de bases de datos deben garantizar la fiabilidad de la información almacenada, a pesar de las caídas del sistema o de los intentos de acceso no autorizados. Si los datos van a ser compartidos entre diferentes usuarios, el sistema debe evitar posibles resultados anómalos.

Dado que la información es tan importante en la mayoría de las organizaciones, los científicos informáticos han desarrollado una gran cuerpo de conceptos y técnicas para la gestión de los datos. Estos conceptos y técnicas constituyen el objetivo central de este libro. En este capítulo se presenta una breve introducción a los principios de los sistemas de bases de datos.

1.1 Aplicaciones de los sistemas de bases de datos

Las bases de datos se usan ampliamente. Algunas de sus aplicaciones representativas son:

- *Banca*: para información de los clientes, cuentas, préstamos y transacciones bancarias.
- *Líneas aéreas*: para reservas e información de horarios. Las líneas aéreas fueron de las primeras en usar las bases de datos de forma distribuida geográficamente.
- *Universidades*: para información de los estudiantes, matrículas en las asignaturas y cursos.
- *Transacciones de tarjetas de crédito*: para compras con tarjeta de crédito y la generación de los extractos mensuales.
- *Telecomunicaciones*: para guardar un registro de las llamadas realizadas, generar las facturas mensuales, mantener el saldo de las tarjetas telefónicas de prepago y para almacenar información sobre las redes de comunicaciones.
- *Finanzas*: para almacenar información sobre compañías tenedoras, ventas y compras de productos financieros, como acciones y bonos; también para almacenar datos del mercado en tiempo real para permitir a los clientes la compraventa en línea y a la compañía la compraventa automática.
- *Ventas*: para información de clientes, productos y compras.

- *Comercio en línea*: para los datos de ventas ya mencionados y para el seguimiento de los pedidos Web, generación de listas de recomendaciones y mantenimiento de evaluaciones de productos en línea.
- *Producción*: para la gestión de la cadena de proveedores y para el seguimiento de la producción de artículos en las factorías, inventarios en los almacenes y pedidos.
- *Recursos humanos*: para información sobre los empleados, salarios, impuestos sobre los sueldos y prestaciones sociales, y para la generación de las nóminas.

Como muestra esta lista, las bases de datos forman una parte esencial de casi todas las empresas actuales.

Durante las últimas cuatro décadas del siglo veinte, el uso de las bases de datos creció en todas las empresas. En los primeros días, muy pocas personas interactuaban directamente con los sistemas de bases de datos, aunque sin darse cuenta interactuaban indirectamente con bases de datos—con informes impresos como los extractos de las tarjetas de crédito, o mediante agentes como los cajeros de los bancos y los agentes de reservas de las líneas aéreas. Después vinieron los cajeros automáticos y permitieron a los usuarios interactuar directamente con las bases de datos. Las interfaces telefónicas con las computadoras (sistemas de respuesta vocal interactiva) también permitieron a los usuarios tratar directamente con las bases de datos—la persona que llamaba podía marcar un número y pulsar las teclas del teléfono para introducir información o para seleccionar opciones alternativas, para conocer las horas de llegada o salida de los vuelos, por ejemplo, o para matricularse de asignaturas en una universidad.

La revolución de Internet a finales de los años noventa aumentó significativamente el acceso directo del usuario a las bases de datos. Las organizaciones convirtieron muchas de sus interfaces telefónicas a las bases de datos en interfaces Web, y dejaron disponibles en línea muchos servicios. Por ejemplo, cuando se accede a una librería en línea y se busca en una colección de libros o de música, se está accediendo a datos almacenados en una base de datos. Cuando se realiza un pedido en línea, el pedido se almacena en una base de datos. Cuando se accede al sitio Web de un banco y se consultan el estado de la cuenta y los movimientos, la información se recupera del sistema de bases de datos del banco. Cuando se accede a un sitio Web, puede que se recupere información personal de una base de datos para seleccionar los anuncios que se deben mostrar. Más aún, los datos sobre los accesos Web pueden almacenarse en una base de datos.

Así, aunque las interfaces de usuario ocultan los detalles del acceso a las bases de datos, y la mayoría de la gente ni siquiera es consciente de que están interactuando con una base de datos, el acceso a las bases de datos forma actualmente una parte esencial de la vida de casi todas las personas.

La importancia de los sistemas de bases de datos se puede juzgar de otra forma—actualmente, los fabricantes de sistemas de bases de datos como Oracle están entre las mayores compañías de software del mundo, y los sistemas de bases de datos forman una parte importante de la línea de productos de compañías más diversificadas como Microsoft e IBM.

1.2 Propósito de los sistemas de bases de datos

Los sistemas de bases de datos surgieron en respuesta a los primeros métodos de gestión informatizada de los datos comerciales. A modo de ejemplo de dichos métodos, típicos de los años sesenta, considérese parte de una entidad bancaria que, entre otros datos, guarda información sobre todos los clientes y todas las cuentas de ahorro. Una manera de guardar la información en la computadora es almacenarla en archivos del sistema operativo. Para permitir que los usuarios manipulen la información, el sistema tiene varios programas de aplicación que gestionan los archivos, incluyendo programas para:

- Efectuar cargos o abonos en las cuentas.
- Añadir cuentas nuevas.
- Calcular el saldo de las cuentas.
- Generar los extractos mensuales.

Estos programas de aplicación los han escrito programadores de sistemas en respuesta a las necesidades del banco.

Se añaden nuevos programas de aplicación al sistema según surgen las necesidades. Por ejemplo, supóngase que una caja de ahorros decide ofrecer cuentas corrientes. En consecuencia, se crean nuevos archivos permanentes que contienen información acerca de todas las cuentas corrientes abiertas en el banco y puede que haya que escribir nuevos programas de aplicación para afrontar situaciones que no se dan en las cuentas de ahorro, como los descubiertos. Así, con el paso del tiempo, se añaden más archivos y programas de aplicación al sistema.

Los sistemas operativos convencionales soportan este **sistema de procesamiento de archivos** típico. El sistema almacena los registros permanentes en varios archivos y necesita diferentes programas de aplicación para extraer y añadir a los archivos correspondientes. Antes de la aparición de los sistemas gestores de bases de datos (SGBDs), las organizaciones normalmente almacenaban la información en sistemas de este tipo.

Guardar la información de la organización en un sistema de procesamiento de archivos tiene una serie de inconvenientes importantes:

- **Redundancia e inconsistencia de los datos.** Debido a que los archivos y programas de aplicación los crean diferentes programadores en el transcurso de un largo período de tiempo, es probable que los diversos archivos tengan estructuras diferentes y que los programas estén escritos en varios lenguajes de programación diferentes. Además, puede que la información esté duplicada en varios lugares (archivos). Por ejemplo, la dirección y el número de teléfono de un cliente dado pueden aparecer en un archivo que contenga registros de cuentas de ahorros y en un archivo que contenga registros de cuentas corrientes. Esta redundancia conduce a costes de almacenamiento y de acceso más elevados. Además, puede dar lugar a la **inconsistencia de los datos**; es decir, puede que las diferentes copias de los mismos datos no coincidan. Por ejemplo, puede que el cambio en la dirección de un cliente esté reflejado en los registros de las cuentas de ahorro pero no en el resto del sistema.
- **Dificultad en el acceso a los datos.** Supóngase que uno de los empleados del banco necesita averiguar los nombres de todos los clientes que viven en un código postal dado. El empleado pide al departamento de procesamiento de datos que genere esa lista. Debido a que esta petición no fue prevista por los diseñadores del sistema original, no hay un programa de aplicación a mano para satisfacerla. Hay, sin embargo, un programa de aplicación que genera la lista de todos los clientes. El empleado del banco tiene ahora dos opciones: bien obtener la lista de *todos* los clientes y extraer manualmente la información que necesita, o bien pedir a un programador de sistemas que escriba el programa de aplicación necesario. Ambas alternativas son obviamente insatisfactorias. Supóngase que se escribe el programa y que, varios días más tarde, el mismo empleado necesita reducir esa lista para que incluya únicamente a aquellos clientes que tengan una cuenta con saldo igual o superior a 10.000 €. Como se puede esperar, no existe ningún programa que genere tal lista. De nuevo, el empleado tiene que elegir entre dos opciones, ninguna de las cuales es satisfactoria.

La cuestión aquí es que los entornos de procesamiento de archivos convencionales no permiten recuperar los datos necesarios de una forma práctica y eficiente. Hacen falta sistemas de recuperación de datos más adecuados para el uso general.

- **Aislamiento de datos.** Como los datos están dispersos en varios archivos, y los archivos pueden estar en diferentes formatos, es difícil escribir nuevos programas de aplicación para recuperar los datos correspondientes.
- **Problemas de integridad.** Los valores de los datos almacenados en la base de datos deben satisfacer ciertos tipos de **restricciones de consistencia**. Por ejemplo, el saldo de ciertos tipos de cuentas bancarias no puede nunca ser inferior a una cantidad predeterminada (por ejemplo, 25 €). Los desarrolladores hacen cumplir esas restricciones en el sistema añadiendo el código correspondiente en los diversos programas de aplicación. Sin embargo, cuando se añaden nuevas restricciones, es difícil cambiar los programas para hacer que se cumplan. El problema se complica cuando las restricciones implican diferentes elementos de datos de diferentes archivos.

- **Problemas de atomicidad.** Los sistemas informáticos, como cualquier otro dispositivo mecánico o eléctrico, está sujeto a fallos. En muchas aplicaciones es crucial asegurar que, si se produce algún fallo, los datos se restauren al estado consistente que existía antes del fallo. Considérese un programa para transferir 50 € desde la cuenta A a la B. Si se produce un fallo del sistema durante la ejecución del programa, es posible que los 50 € fueran retirados de la cuenta A pero no abonados en la cuenta B, dando lugar a un estado inconsistente de la base de datos. Evidentemente, resulta esencial para la consistencia de la base de datos que tengan lugar tanto el abono como el cargo, o que no tenga lugar ninguno. Es decir, la transferencia de fondos debe ser *atómica*—debe ocurrir en su totalidad o no ocurrir en absoluto. Resulta difícil asegurar la atomicidad en los sistemas convencionales de procesamiento de archivos.
- **Anomalías en el acceso concurrente.** Para aumentar el rendimiento global del sistema y obtener una respuesta más rápida, muchos sistemas permiten que varios usuarios actualicen los datos simultáneamente. En realidad, hoy en día, los principales sitios de comercio electrónico de Internet pueden tener millones de accesos diarios de compradores a sus datos. En tales entornos es posible la interacción de actualizaciones concurrentes y puede dar lugar a datos inconsistentes. Considérese una cuenta bancaria A, que contenga 500 €. Si dos clientes retiran fondos (por ejemplo, 50 € y 100 €, respectivamente) de la cuenta A aproximadamente al mismo tiempo, el resultado de las ejecuciones concurrentes puede dejar la cuenta en un estado incorrecto (o inconsistente). Supóngase que los programas que se ejecutan para cada retirada leen el saldo anterior, reducen su valor en el importe que se retira y luego escriben el resultado. Si los dos programas se ejecutan concurrentemente, pueden leer el valor 500 €, y escribir después 450 € y 400 €, respectivamente. Dependiendo de cuál escriba el valor en último lugar, la cuenta puede contener 450 € o 400 €, en lugar del valor correcto, 350 €. Para protegerse contra esta posibilidad, el sistema debe mantener alguna forma de supervisión. Pero es difícil ofrecer supervisión, ya que muchos programas de aplicación diferentes que no se han coordinado con anterioridad pueden tener acceso a los datos.
- **Problemas de seguridad.** No todos los usuarios de un sistema de bases de datos deben poder acceder a todos los datos. Por ejemplo, en un sistema bancario, el personal de nóminas sólo necesita ver la parte de la base de datos que contiene información acerca de los diferentes empleados del banco. No necesitan tener acceso a la información acerca de las cuentas de clientes. Pero, como los programas de aplicación se añaden al sistema de procesamiento de datos de una forma ad hoc, es difícil hacer cumplir tales restricciones de seguridad.

Estas dificultades, entre otras, motivaron el desarrollo de los sistemas de bases de datos. En el resto del libro se examinarán los conceptos y los algoritmos que permiten que los sistemas de bases de datos resuelvan los problemas de los sistemas de procesamiento de archivos. En general, en este libro se usa una entidad bancaria como ejemplo de aplicación típica de procesamiento de datos que puede encontrarse en una empresa.

1.3 Visión de los datos

Un sistema de bases de datos es una colección de datos interrelacionados y un conjunto de programas que permiten a los usuarios tener acceso a esos datos y modificarlos. Una de las principales finalidades de los sistemas de bases de datos es ofrecer a los usuarios una visión *abstracta* de los datos. Es decir, el sistema oculta ciertos detalles del modo en que se almacenan y mantienen los datos.

1.3.1 Abstracción de datos

Para que el sistema sea útil debe recuperar los datos eficientemente. La necesidad de eficiencia ha llevado a los diseñadores a usar estructuras de datos complejas para la representación de los datos en la base de datos. Dado que muchos de los usuarios de sistemas de bases de datos no tienen formación en informática, los desarrolladores ocultan esa complejidad a los usuarios mediante varios niveles de abstracción para simplificar la interacción de los usuarios con el sistema:

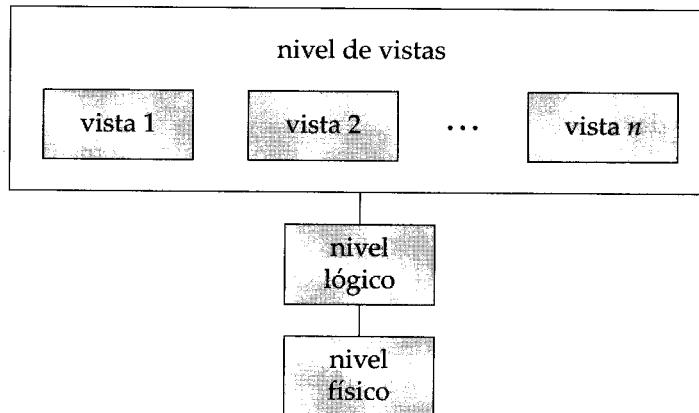


Figura 1.1 Los tres niveles de abstracción de datos.

- **Nivel físico.** El nivel más bajo de abstracción describe *cómo* se almacenan realmente los datos. El nivel físico describe en detalle las estructuras de datos complejas de bajo nivel.
- **Nivel lógico.** El nivel inmediatamente superior de abstracción describe *qué* datos se almacenan en la base de datos y qué relaciones existen entre esos datos. El nivel lógico, por tanto, describe toda la base de datos en términos de un número pequeño de estructuras relativamente simples. Aunque la implementación de esas estructuras simples en el nivel lógico puede involucrar estructuras complejas del nivel físico, los usuarios del nivel lógico no necesitan preocuparse de esta complejidad. Los administradores de bases de datos, que deben decidir la información que se guarda en la base de datos, usan el nivel de abstracción lógico.
- **Nivel de vistas.** El nivel más elevado de abstracción sólo describe parte de la base de datos. Aunque el nivel lógico usa estructuras más simples, queda algo de complejidad debido a la variedad de información almacenada en las grandes bases de datos. Muchos usuarios del sistema de bases de datos no necesitan toda esta información; en su lugar sólo necesitan tener acceso a una parte de la base de datos. El nivel de abstracción de vistas existe para simplificar su interacción con el sistema. El sistema puede proporcionar muchas vistas para la misma base de datos.

La Figura 1.1 muestra la relación entre los tres niveles de abstracción.

Una analogía con el concepto de tipos de datos en lenguajes de programación puede clarificar la diferencia entre los niveles de abstracción. La mayoría de los lenguajes de programación de alto nivel soportan el concepto de tipo estructurado. Por ejemplo, en los lenguajes tipo Pascal se pueden declarar registros de la manera siguiente:

```

type cliente = record
    id_cliente : string;
    nombre_cliente : string;
    calle_cliente : string;
    ciudad_cliente : string;
end;
  
```

Este código define un nuevo tipo de registro denominado *cliente* con cuatro campos. Cada campo tiene un nombre y un tipo asociados. Una entidad bancaria puede tener varios tipos de estos registros, incluidos:

- *cuenta*, con los campos *número_cuenta* y *saldo*.
- *empleado*, con los campos *nombre_empleado* y *sueldo*.

En el nivel físico, los registros *cliente*, *cuenta* o *empleado* se pueden describir como bloques de posiciones consecutivas de almacenamiento (por ejemplo, palabras o bytes). El compilador oculta este nivel

de detalle a los programadores. De manera parecida, el sistema de base de datos oculta muchos de los detalles de almacenamiento de los niveles inferiores a los programadores de bases de datos. Los administradores de bases de datos, por otro lado, pueden ser conscientes de ciertos detalles de la organización física de los datos.

En el nivel lógico cada registro de este tipo se describe mediante una definición de tipo, como en el fragmento de código anterior, y también se define la relación entre estos tipos de registros. Los programadores que usan un lenguaje de programación trabajan en este nivel de abstracción. De manera parecida, los administradores de bases de datos suelen trabajar en este nivel de abstracción.

Finalmente, en el nivel de vistas, los usuarios de computadoras ven un conjunto de programas de aplicación que ocultan los detalles de los tipos de datos. De manera parecida, en el nivel de vistas se definen varias vistas de la base de datos y los usuarios de la base de datos pueden verlas. Además de ocultar los detalles del nivel lógico de la base de datos, las vistas también proporcionan un mecanismo de seguridad para evitar que los usuarios tengan acceso a ciertas partes de la base de datos. Por ejemplo, los cajeros de un banco sólo ven la parte de la base de datos que contiene información de las cuentas de los clientes; no pueden tener acceso a la información referente a los sueldos de los empleados.

1.3.2 Ejemplares y esquemas

Las bases de datos van cambiando a lo largo del tiempo conforme la información se inserta y se elimina. La colección de información almacenada en la base de datos en un momento dado se denomina **ejemplar** de la base de datos. El diseño general de la base de datos se denomina **esquema** de la base de datos. Los esquemas se modifican rara vez, si es que se modifican.

El concepto de esquemas y ejemplares de las bases de datos se puede comprender por analogía con los programas escritos en un lenguaje de programación. El esquema de la base de datos se corresponde con las declaraciones de las variables (junto con las definiciones de tipos asociadas) de los programas. Cada variable tiene un valor concreto en un instante dado. Los valores de las variables de un programa en un instante dado se corresponden con un *ejemplar* del esquema de la base de datos.

Los sistemas de bases de datos tienen varios esquemas divididos según los niveles de abstracción. El **esquema físico** describe el diseño de la base de datos en el nivel físico, mientras que el **esquema lógico** describe su diseño en el nivel lógico. Las bases de datos también pueden tener varios esquemas en el nivel de vistas, a veces denominados **subesquemas**, que describen diferentes vistas de la base de datos.

De éstos, el esquema lógico es con mucho el más importante en términos de su efecto sobre los programas de aplicación, ya que los programadores crean las aplicaciones usando el esquema lógico. El esquema físico está oculto bajo el esquema lógico, y generalmente puede modificarse fácilmente sin afectar a los programas de aplicación. Se dice que los programas de aplicación muestran **independencia física respecto de los datos** si no dependen del esquema físico y, por tanto, no hace falta volver a escribirlos si se modifica el esquema físico.

Se estudiarán los lenguajes para la descripción de los esquemas, después de introducir el concepto de modelos de datos en el apartado siguiente.

1.3.3 Modelos de datos

Bajo la estructura de las bases de datos se encuentra el **modelo de datos**: una colección de herramientas conceptuales para describir los datos, sus relaciones, su semántica y las restricciones de consistencia. Los modelos de datos ofrecen un modo de describir el diseño de las bases de datos en los niveles físico, lógico y de vistas.

En este texto se van a tratar varios modelos de datos diferentes. Los modelos de datos pueden clasificarse en cuatro categorías diferentes:

- **Modelo relacional.** El modelo relacional usa una colección de tablas para representar tanto los datos como sus relaciones. Cada tabla tiene varias columnas, y cada columna tiene un nombre único. El modelo relacional es un ejemplo de un modelo basado en registros. Los modelos basados en registros se denominan así porque la base de datos se estructura en registros de formato fijo de varios tipos. Cada tabla contiene registros de un tipo dado. Cada tipo de registro define un número fijo de campos, o atributos. Las columnas de la tabla se corresponden con los atributos

del tipo de registro. El modelo de datos relacional es el modelo de datos más ampliamente usado, y una gran mayoría de sistemas de bases de datos actuales se basan en el modelo relacional. Los Capítulos 2 al 7 tratan el modelo relacional en detalle.

- **El modelo entidad-relación.** El modelo de datos entidad-relación (E-R) se basa en una percepción del mundo real que consiste en una colección de objetos básicos, denominados *entidades*, y de las *relaciones* entre ellos. Una entidad es una “cosa” u “objeto” del mundo real que es distinguible de otros objetos. El modelo entidad-relación se usa mucho en el diseño de bases de datos y en el Capítulo 6 se examina detalladamente.
- **Modelo de datos orientado a objetos.** El modelo de datos orientado a objetos es otro modelo de datos que está recibiendo una atención creciente. El modelo orientado a objetos se puede considerar como una extensión del modelo E-R con los conceptos de la encapsulación, los métodos (funciones) y la identidad de los objetos. En el Capítulo 9 se examina este modelo de datos.
- **Modelo de datos semiestructurados.** El modelo de datos semiestructurados permite la especificación de datos donde los elementos de datos individuales del mismo tipo pueden tener diferentes conjuntos de atributos. Esto lo diferencia de los modelos de datos mencionados anteriormente, en los que cada elemento de datos de un tipo particular debe tener el mismo conjunto de atributos. El **lenguaje de marcas extensible** (XML, eXtensible Markup Language) se emplea mucho para representar datos semiestructurados. Se estudia en el Capítulo 10.

El **modelo de datos de red** y el **modelo de datos jerárquico** precedieron cronológicamente al relacional. Estos modelos estuvieron íntimamente ligados a la implementación subyacente y complicaban la tarea del modelado de datos. En consecuencia, se usan muy poco hoy en día, excepto en el código de bases de datos antiguas que sigue estando en servicio en algunos lugares. Se describen brevemente en los Apéndices A y B para los lectores interesados.

1.4 Lenguajes de bases de datos

Los sistemas de bases de datos proporcionan un **lenguaje de definición de datos** para especificar el esquema de la base de datos y un **lenguaje de manipulación de datos** para expresar las consultas y las modificaciones de la base de datos. En la práctica, los lenguajes de definición y manipulación de datos no son dos lenguajes diferentes; en cambio, simplemente forman parte de un único lenguaje de bases de datos, como puede ser el muy usado SQL.

1.4.1 Lenguaje de manipulación de datos

Un **lenguaje de manipulación de datos** (LMD) es un lenguaje que permite a los usuarios tener acceso a los datos organizados mediante el modelo de datos correspondiente o manipularlos. Los tipos de acceso son:

- La recuperación de la información almacenada en la base de datos.
- La inserción de información nueva en la base de datos.
- El borrado de la información de la base de datos.
- La modificación de la información almacenada en la base de datos.

Hay fundamentalmente dos tipos:

- Los **LMDs procedimentales** necesitan que el usuario especifique *qué* datos se necesitan y *cómo* obtener esos datos.
- Los **LMDs declarativos** (también conocidos como **LMDs no procedimentales**) necesitan que el usuario especifique *qué* datos se necesitan *sin* que haga falta que especifique cómo obtener esos datos.

Los LMDs declarativos suelen resultar más fáciles de aprender y de usar que los procedimentales. Sin embargo, como el usuario no tiene que especificar cómo conseguir los datos, el sistema de bases de datos tiene que determinar un medio eficiente de acceso a los datos.

Una **consulta** es una instrucción que solicita que se recupere información. La parte de los LMDs implicada en la recuperación de información se denomina **lenguaje de consultas**. Aunque técnicamente sea incorrecto, resulta habitual usar las expresiones *lenguaje de consultas* y *lenguaje de manipulación de datos* como sinónimas.

Existen varios lenguajes de consultas de bases de datos en uso, tanto comercial como experimentalmente. El lenguaje de consultas más ampliamente usado, SQL, se estudiará en los Capítulos 3 y 4. También se estudiarán otros lenguajes de consultas en el Capítulo 5.

Los niveles de abstracción que se trataron en el Apartado 1.3 no sólo se aplican a la definición o estructuración de datos, sino también a su manipulación. En el nivel físico se deben definir los algoritmos que permitan un acceso eficiente a los datos. En los niveles superiores de abstracción se pone el énfasis en la facilidad de uso. El objetivo es permitir que los seres humanos interactúen de manera eficiente con el sistema. El componente procesador de consultas del sistema de bases de datos (que se estudia en los Capítulos 13 y 14) traduce las consultas LMD en secuencias de acciones en el nivel físico del sistema de bases de datos.

1.4.2 Lenguaje de definición de datos

Los esquemas de las bases de datos se especifican mediante un conjunto de definiciones expresadas mediante un lenguaje especial denominado **lenguaje de definición de datos (LDD)**. El LDD también se usa para especificar más propiedades de los datos.

La estructura de almacenamiento y los métodos de acceso usados por el sistema de bases de datos se especifican mediante un conjunto de instrucciones en un tipo especial de LDD denominado **lenguaje de almacenamiento y definición de datos**. Estas instrucciones definen los detalles de implementación de los esquemas de las bases de datos, que suelen ocultarse a los usuarios.

Los valores de los datos almacenados en la base de datos deben satisfacer ciertas **restricciones de consistencia**. Por ejemplo, supóngase que el saldo de una cuenta no debe caer por debajo de 100 €. El LDD proporciona facilidades para especificar tales restricciones. Los sistemas de bases de datos las comproban cada vez que se modifica la base de datos. En general, las restricciones pueden ser predicados arbitrarios relativos a la base de datos. No obstante, los predicados arbitrarios pueden resultar costosos de comprobar. Por tanto, los sistemas de bases de datos se concentran en las restricciones de integridad que pueden comprobarse con una sobrecarga mínima:

- **Restricciones de dominio.** Se debe asociar un dominio de valores posibles a cada atributo (por ejemplo, tipos enteros, tipos de carácter, tipos fecha/hora). La declaración de un atributo como parte de un dominio concreto actúa como restricción de los valores que puede adoptar. Las restricciones de dominio son la forma más elemental de restricción de integridad. El sistema las comprueba fácilmente siempre que se introduce un nuevo elemento de datos en la base de datos.
- **Integridad referencial.** Hay casos en los que se desea asegurar que un valor que aparece en una relación para un conjunto de atributos dado aparece también para un determinado conjunto de atributos en otra relación (integridad referencial). Las modificaciones de la base de datos pueden causar violaciones de la integridad referencial. Cuando se viola una restricción de integridad, el procedimiento normal es rechazar la acción que ha causado esa violación.
- **Asertos.** Un aserto es cualquier condición que la base de datos debe satisfacer siempre. Las restricciones de dominio y las restricciones de integridad referencial son formas especiales de asertos. No obstante, hay muchas restricciones que no pueden expresarse empleando únicamente esas formas especiales. Por ejemplo: "Cada préstamo tiene como mínimo un cliente tenedor de una cuenta con un saldo mínimo de 1.000,00 €" debe expresarse en forma de aserto. Cuando se crea un aserto, el sistema comprueba su validez. Si el aserto es válido, cualquier modificación futura de la base de datos se permite únicamente si no hace que se viole ese aserto.
- **Autorización.** Puede que se desee diferenciar entre los usuarios en cuanto al tipo de acceso que se les permite a diferentes valores de los datos de la base de datos. Estas diferenciaciones se

expresan en términos de **autorización**, cuyas modalidades más frecuentes son: **autorización de lectura**, que permite la lectura pero no la modificación de los datos; **autorización de inserción**, que permite la inserción de datos nuevos, pero no la modificación de los datos ya existentes; **autorización de actualización**, que permite la modificación, pero no la eliminación, de los datos; y la **autorización de eliminación**, que permite la eliminación de datos. A cada usuario se le pueden asignar todos, ninguno o una combinación de estos tipos de autorización.

El LDD, al igual que cualquier otro lenguaje de programación, obtiene como entrada algunas instrucciones y genera una salida. La salida del LDD se coloca en el **diccionario de datos**, que contiene **metadatos**—es decir, datos sobre datos. El diccionario de datos se considera un tipo especial de tabla, a la que sólo puede tener acceso y actualizar el propio sistema de bases de datos (no los usuarios normales). El sistema de bases de datos consulta el diccionario de datos antes de leer o modificar los datos reales.

1.5 Bases de datos relacionales

Las bases de datos relacionales se basan en el modelo relacional y usan un conjunto de tablas para representar tanto los datos como las relaciones entre ellos. También incluyen un LMD y un LDD. La mayor parte de los sistemas de bases de datos relacionales comerciales emplean el lenguaje SQL, que se trata en este apartado y que se tratará con gran detalle en los Capítulos 3 y 4. En el Capítulo 5 se estudiarán otros lenguajes influyentes.

1.5.1 Tablas

Cada tabla tiene varias columnas, y cada columna tiene un nombre único. En la Figura 1.2 se presenta un ejemplo de base de datos relacional consistente en tres tablas: una muestra detalles de los clientes de un banco, la segunda muestra las cuentas y la tercera muestra las cuentas que pertenecen a cada cliente.

La primera tabla, la tabla *cliente*, muestra, por ejemplo, que el cliente identificado por *id_cliente* 19.283.746 se llama González y vive en la calle Arenal en La Granja. La segunda tabla, *cuenta*, muestra, por ejemplo, que la cuenta C-101 tiene un saldo de 500 € y la C-201 un saldo de 900 €.

La tercera tabla muestra las cuentas que pertenecen a cada cliente. Por ejemplo, la cuenta C-101 pertenece al cliente cuyo *id_cliente* es 19.283.746 (González), y los clientes 19.283.746 (González) y 01.928.374 (Gómez) comparten el número de cuenta C-201 (pueden compartir un negocio).

El modelo relacional es un ejemplo de modelo basado en registros. Los modelos basados en registros se denominan así porque la base de datos se estructura en registros de formato fijo de varios tipos. Cada tabla contiene registros de un tipo dado. Cada tipo de registro define un número fijo de campos, o atributos. Las columnas de la tabla se corresponden con los atributos del tipo de registro.

No es difícil ver cómo se pueden almacenar las tablas en archivos. Por ejemplo, se puede usar un carácter especial (como la coma) para delimitar los diferentes atributos de un registro, y otro carácter especial (como el carácter de nueva línea) para delimitar los registros. El modelo relacional oculta esos detalles de implementación de bajo nivel a los desarrolladores de bases de datos y a los usuarios.

El modelo de datos relacional es el modelo de datos más ampliamente usado, y una gran mayoría de los sistemas de bases de datos actuales se basan en el modelo relacional. Los Capítulos 2 al 7 tratan el modelo relacional con detalle.

Obsérvese también que en el modelo relacional es posible crear esquemas que tengan problemas tales como información duplicada innecesariamente. Por ejemplo, supóngase que se almacena *número_cuenta* como atributo de un registro *cliente*. Entonces, para representar el hecho de que tanto la cuenta C-101 como la cuenta C-201 pertenece al cliente González (con *id_cliente* 19.283.746) sería necesario almacenar dos filas en la tabla *cliente*. Los valores de *nombre_cliente*, *calle_cliente* y *ciudad_cliente* de González estarían innecesariamente duplicados en las dos filas. En el Capítulo 7 se estudiará el modo de distinguir los buenos diseños de esquema de los malos.

<i>id_cliente</i>	<i>nombre_cliente</i>	<i>calle_cliente</i>	<i>ciudad_cliente</i>
19.283.746	González	Arenal, 12	La Granja
67.789.901	López	Mayor, 3	Peguerinos
18.273.609	Abril	Preciados, 123	Valsaín
32.112.312	Santos	Mayor, 100	Peguerinos
33.666.999	Rupérez	Ramblas, 175	León
01.928.374	Gómez	Carretas, 72	Cerceda

(a) La tabla *cliente*

<i>número_cuenta</i>	<i>saldo</i>
C-101	500
C-215	700
C-102	400
C-305	350
C-201	900
C-217	750
C-222	700

(b) La tabla *cuenta*

<i>id_cliente</i>	<i>número_cuenta</i>
19.283.746	C-101
19.283.746	C-201
01.928.374	C-215
67.789.901	C-102
18.273.609	C-305
32.112.312	C-217
33.666.999	C-222
01.928.374	C-201

(c) La tabla *impositor***Figura 1.2** Un ejemplo de base de datos relacional.

1.5.2 Lenguaje de manipulación de datos

El lenguaje de consultas de SQL no es procedimental. Usa como entrada varias tablas (posiblemente sólo una) y devuelve siempre una sola tabla. A continuación se ofrece un ejemplo de consulta SQL que halla el nombre de todos los clientes que residen en Peguerinos:

```
select cliente.nombre_cliente
from cliente
where cliente.ciudad_cliente = 'Peguerinos'
```

La consulta especifica que hay que recuperar (*select*) las filas de (*from*) la tabla *cliente* en las que (*where*) la *ciudad_cliente* es Peguerinos, y que sólo debe mostrarse el atributo *nombre_cliente* de esas filas. Más concretamente, el resultado de la ejecución de esta consulta es una tabla con una sola columna denominada *nombre_cliente* y un conjunto de filas, cada una de las cuales contiene el nombre de un cliente cuya *ciudad_cliente* es Peguerinos. Si la consulta se ejecuta sobre la tabla de la Figura 1.2, el resultado constará de dos filas, una con el nombre López y otra con el nombre Santos.

Las consultas pueden involucrar información de más de una tabla. Por ejemplo, la siguiente consulta busca todos los números de cuenta y sus saldos del cliente con *id_cliente* 19.283.746.

```
select cuenta.número_cuenta, cuenta.saldo
from impositor, cuenta
where impositor.id_cliente = '19.283.746' and
      impositor.número_cuenta = cuenta.número_cuenta
```

Si la consulta anterior se ejecutase sobre las tablas de la Figura 1.2, el sistema encontraría que las dos cuentas denominadas C-101 y C-201 pertenecen al cliente 19.283.746 y el resultado consistiría en una tabla con dos columnas (*número_cuenta*, *saldo*) y dos filas (C-101, 500) y (C-201, 900).

1.5.3 Lenguaje de definición de datos

SQL ofrece un LDD elaborado que permite definir tablas, restricciones de integridad, asertos, etc.

Por ejemplo, la siguiente instrucción del lenguaje SQL define la tabla *cuenta*:

```
create table cuenta
(número_cuenta char(10),
 saldo integer)
```

La ejecución de esta instrucción LDD crea la tabla *cuenta*. Además, actualiza el diccionario de datos, que contiene metadatos (1.4.2). Los esquemas de las tablas son ejemplos de metadatos.

1.5.4 Acceso a las bases de datos desde los programas de aplicación

SQL no es tan potente como la máquina universal de Turing; es decir, hay algunos cálculos que no pueden obtenerse mediante ninguna consulta SQL. Esos cálculos deben escribirse en un lenguaje anfitrión, como Cobol, C, C++ o Java. Los **programas de aplicación** son programas que se usan para interactuar de esta manera con las bases de datos. Algunos de los ejemplos de un sistema bancario serían los programas que generan las nóminas, realizan cargos en las cuentas, realizan abonos en las cuentas o transfieren fondos entre las cuentas.

Para tener acceso a la base de datos, las instrucciones LMD deben ejecutarse desde el lenguaje anfitrión. Hay dos maneras de conseguirlo:

- Proporcionando una interfaz de programas de aplicación (conjunto de procedimientos) que se pueda usar para enviar instrucciones LMD y LDD a la base de datos y recuperar los resultados.
El estándar de conectividad abierta de bases de datos (ODBC, Open Data Base Connectivity) definido por Microsoft para su empleo con el lenguaje C es un estándar de interfaz de programas de aplicación usado habitualmente. El estándar de conectividad de Java con bases de datos (JDBC, Java Data Base Connectivity) ofrece las características correspondientes para el lenguaje Java.
- Extendiendo la sintaxis del lenguaje anfitrión para que incorpore las llamadas LMD dentro del programa del lenguaje anfitrión. Generalmente, un carácter especial precede a las llamadas LMD y un preprocesador, denominado **precompilador LMD**, convierte las instrucciones LMD en llamadas normales a procedimientos en el lenguaje anfitrión.

1.6 Diseño de bases de datos

Los sistemas de bases de datos se diseñan para gestionar grandes cantidades de información. Esas grandes cantidades de información no existen aisladas. Forman parte del funcionamiento de alguna empresa, cuyo producto final puede que sea la información obtenida de la base de datos o algún dispositivo o servicio para el que la base de datos sólo desempeña un papel secundario.

El diseño de bases de datos implica principalmente el diseño del esquema de las bases de datos. El diseño de un entorno completo de aplicaciones para la base de datos que satisfaga las necesidades de la empresa que se está modelando exige prestar atención a un conjunto de aspectos más amplio. Este texto se centrará inicialmente en la escritura de las consultas a la base de datos y en el diseño de los esquemas de las bases de datos. En el Capítulo 8 se estudia el proceso general de diseño de las aplicaciones.

1.6.1 Proceso de diseño

Los modelos de datos de alto nivel resultan útiles a los diseñadores de bases de datos al ofrecerles un marco conceptual en el que especificar, de manera sistemática, los requisitos de datos de los usuarios de las bases de datos y la manera en que se estructurará la base de datos para satisfacer esos requisitos. La fase inicial del diseño de las bases de datos, por tanto, es caracterizar completamente los requisitos de datos de los hipotéticos usuarios de la base de datos. Los diseñadores de bases de datos deben interactuar ampliamente con los expertos y usuarios del dominio para llevar a cabo esta tarea. El resultado de esta fase es la especificación de los requisitos de los usuarios.

A continuación, el diseñador escoge un modelo de datos y, mediante la aplicación de los conceptos del modelo de datos elegido, traduce esos requisitos en un esquema conceptual de la base de datos. El esquema desarrollado en esta fase de **diseño conceptual** ofrece una visión general detallada de la empresa. El diseñador revisa el esquema para confirmar que todos los requisitos de datos se satisfacen realmente y no entran en conflicto entre sí. El diseñador también puede examinar el diseño para eliminar cualquier característica redundante. En este punto, la atención se centra en describir los datos y sus relaciones, más que en especificar los detalles del almacenamiento físico.

En términos del modelo relacional, el proceso de diseño conceptual implica decisiones sobre *qué* atributos se desea capturar en la base de datos y *cómo agruparlos* para formar las diferentes tablas. La parte “*qué*” es, esencialmente, una decisión conceptual, y no se seguirá estudiando en este texto. La parte del “*cómo*” es, esencialmente, un problema informático. Hay dos vías principales para afrontar el problema. La primera supone usar el modelo entidad-relación (Apartado 1.6.3); la otra es emplear un conjunto de algoritmos (denominados colectivamente como *normalización*) que toma como entrada el conjunto de todos los atributos y genera un conjunto de tablas (Apartado 1.6.4).

Un esquema conceptual completamente desarrollado también indica los requisitos funcionales de la empresa. En la **especificación de requisitos funcionales** los usuarios describen el tipo de operaciones (o transacciones) que se llevarán a cabo con los datos. Un ejemplo de estas operaciones es modificar o actualizar los datos, buscar y recuperar datos concretos y eliminar datos. En esta etapa del diseño conceptual el diseñador puede revisar el esquema para asegurarse de que satisface los requisitos funcionales.

El proceso de pasar de un modelo de datos abstracto a la implementación de la base de datos continúa con dos fases de diseño finales. En la **fase de diseño lógico** el diseñador relaciona el esquema conceptual de alto nivel con el modelo de implementación de datos del sistema de bases de datos que se va a usar. El diseñador usa el esquema de bases de datos específico para el sistema resultante en la **fase de diseño físico** posterior, en la que se especifican las características físicas de la base de datos. Entre esas características están la forma de organización de los archivos y las estructuras de almacenamiento interno; se estudian en el Capítulo 11.

1.6.2 Diseño de la base de datos para una entidad bancaria

Para ilustrar el proceso de diseño, examinemos el modo en que puede diseñarse una base de datos para una entidad bancaria. La especificación inicial de los requisitos de los usuarios puede basarse en entrevistas con los usuarios de la base de datos y en el análisis de la entidad realizado por el propio diseñador. La descripción que surge de esta fase de diseño sirve como base para especificar la estructura conceptual de la base de datos. Éstas son las principales características de la entidad bancaria:

- El banco está organizado en sucursales. Cada sucursal se ubica en una ciudad determinada y queda identificada por un nombre único. El banco supervisa los activos de cada sucursal.
- Los clientes quedan identificados por el valor de su *id_cliente*. El banco almacena el nombre de cada cliente y la calle y la ciudad en la que vive. Los clientes pueden abrir cuentas y solicitar préstamos. Cada cliente puede asociarse con un empleado del banco en concreto, que puede actuar como prestamista o como asesor personal para ese cliente.
- El banco ofrece dos tipos de cuenta: de ahorro y corriente. Las cuentas pueden tener como titular a más de un cliente, y cada cliente puede abrir más de una cuenta. A cada cuenta se le asigna un número de cuenta único. El banco guarda un registro del saldo de cada cuenta y de la fecha

más reciente en que cada cliente titular de esa cuenta tuvo acceso a ella. Además, cada cuenta de ahorro tiene una tasa de interés y se registran los descubiertos de las cuentas corrientes.

- El banco ofrece préstamos a sus clientes. Cada préstamo se origina en una sucursal concreta y puede tener como titulares a uno o más clientes. Cada préstamo queda identificado por un número de préstamo único. De cada préstamo el banco realiza un seguimiento del importe del préstamo y de sus pagos. Aunque el número de pago del préstamo no identifica de manera única un pago concreto entre todos los del banco, el número de pago identifica cada pago concreto de un préstamo dado. Se registran la fecha y el importe de cada pago.
- Los empleados del banco quedan identificados por el valor de su *id_empleado*. La administración del banco almacena el nombre y número de teléfono de cada empleado, el nombre de las personas dependientes de cada empleado y el número de *id_empleado* del jefe de cada empleado. El banco también realiza un seguimiento de la fecha de contratación y, por tanto, de la antigüedad de cada empleado.

En las entidades bancarias reales, el banco realizaría un seguimiento de los depósitos y de los reintegros de las cuentas de ahorro y corrientes, igual que realiza un seguimiento de los pagos de las cuentas de crédito. Dado que los requisitos de modelado para ese seguimiento son parecidas y nos gustaría que la aplicación de ejemplo no tuviera un tamaño excesivo, en nuestro modelo no se realiza el seguimiento de esos depósitos y reintegros.

1.6.3 El modelo entidad-relación

El modelo de datos entidad-relación (E-R) está basado en una percepción del mundo real que consiste en un conjunto de objetos básicos, denominados *entidades*, y de las *relaciones* entre esos objetos. Una entidad es una “cosa” u “objeto” del mundo real que es distingible de otros objetos. Por ejemplo, cada persona es una entidad, y las cuentas bancarias pueden considerarse entidades.

Las entidades se describen en las bases de datos mediante un conjunto de **atributos**. Por ejemplo, los atributos *número_cuenta* y *saldo* pueden describir una cuenta concreta de un banco y constituyen atributos del conjunto de entidades *cuenta*. Análogamente, los atributos *nombre_cliente*, *calle_cliente* y *ciudad_cliente* pueden describir una entidad *cliente*.

Se usa un atributo extra, *id_cliente*, para identificar únicamente a los clientes (dado que es posible que haya dos clientes con el mismo nombre, calle y ciudad). Se debe asignar un identificador de cliente único a cada cliente. En Estados Unidos, muchas empresas usan el número de la seguridad social de cada persona (un número único que el Gobierno de Estados Unidos asigna a cada persona) como identificador de cliente.

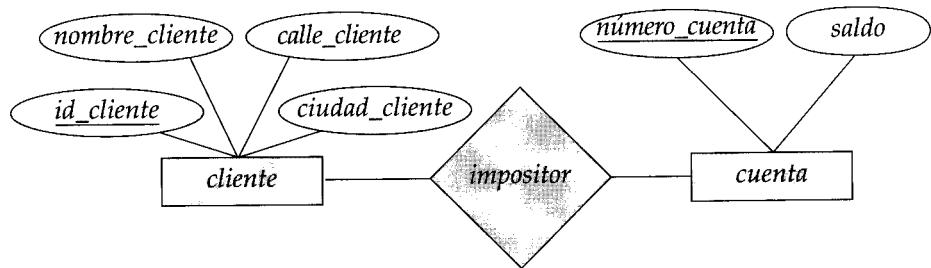
Una **relación** es una asociación entre varias entidades. Por ejemplo, la relación *impositor* asocia un cliente con cada cuenta que tiene. El conjunto de todas las entidades del mismo tipo, y el conjunto de todas las relaciones del mismo tipo se denominan, respectivamente, **conjunto de entidades** y **conjunto de relaciones**.

La estructura lógica general (esquema) de la base de datos se puede expresar gráficamente mediante un *diagrama E-R*, que está constituido por los siguientes componentes:

- **Rectángulos**, que representan conjuntos de entidades.
- **Elipses**, que representan atributos.
- **Rombos**, que representan conjuntos de relaciones entre miembros de varios conjuntos de entidades.
- **Líneas**, que unen los atributos con los conjuntos de entidades entre sí, y también los conjuntos de entidades con las relaciones.

Cada componente se etiqueta con la entidad o relación que representa.

Como ilustración, considérese parte de un sistema bancario de bases de datos consistente en los clientes y las cuentas que tienen esos clientes. La Figura 1.3 muestra el diagrama E-R correspondiente. El

**Figura 1.3** Un ejemplo de diagrama E-R.

<i>id_cliente</i>	<i>número_cuenta</i>	<i>saldo</i>
19.283.746	C-101	500
19.283.746	C-201	900
01.928.374	C-215	700
67.789.901	C-102	400
18.273.609	C-305	350
32.112.312	C-217	750
33.666.999	C-222	700
01.928.374	C-201	900

Figura 1.4 La tabla *impositor'*.

diagrama E-R indica que hay dos conjuntos de entidades, *cliente* y *cuenta*, con los atributos descritos anteriormente. El diagrama también muestra la relación *impositor* entre cliente y cuenta.

Además de entidades y relaciones, el modelo E-R representa ciertas restricciones que los contenidos de la base de datos deben cumplir. Una restricción importante es la **correspondencia de cardinalidades**, que expresa el número de entidades con las que otra entidad se puede asociar a través de un conjunto de relaciones. Por ejemplo, si cada cuenta sólo debe pertenecer a un cliente, el modelo puede expresar esa restricción.

El modelo entidad-relación se usa ampliamente en el diseño de bases de datos, y en el Capítulo 6 se explora en detalle.

1.6.4 Normalización

Otro método de diseño de bases de datos es usar un proceso que suele denominarse normalización. El objetivo es generar un conjunto de esquemas de relaciones que permita almacenar información sin redundancias innecesarias, pero que también permita recuperar la información con facilidad. El enfoque es diseñar esquemas que se hallen en la *forma normal* adecuada. Para determinar si un esquema de relación se halla en una de las formas normales deseadas, hace falta información adicional sobre la empresa real que se está modelando con la base de datos. El enfoque más frecuente es usar **dependencias funcionales**, que se tratan en el Apartado 7.4.

Para comprender la necesidad de la normalización, obsérvese lo que puede fallar en un mal diseño de base de datos. Algunas de las propiedades no deseables de un mal son:

- Repetición de la información.
- Imposibilidad de representar determinada información.

Se examinarán estos problemas con la ayuda de un diseño de bases de datos modificado para el ejemplo bancario.

Supóngase que, en lugar de tener las dos tablas separadas *cuenta* e *impositor*, se tiene una sola tabla, *impositor'*, que combina la información de las dos tablas (como puede verse en la Figura 1.4). Obsérvese que hay dos filas de *impositor'* que contienen información sobre la cuenta C-201. La repetición de información en este diseño alternativo no es deseable. La repetición de información malgasta espacio.

<i>id_cliente</i>	<i>nombre_cliente</i>	<i>calle_cliente</i>	<i>ciudad_cliente</i>	<i>número_cuenta</i>
19.283.746	González	Arenal, 12	La Granja	C-101
19.283.746	González	Arenal, 12	La Granja	C-201
67.789.901	López	Mayor, 3	Peguerinos	C-102
18.273.609	Abril	Preciados, 123	Valsaín	C-305
32.112.312	Santos	Mayor, 100	Peguerinos	C-217
33.666.999	Rupérez	Ramblas, 175	León	C-222
01.928.374	Gómez	Carretas, 72	Cerceda	C-201

Figura 1.5 La tabla *cliente'*.

Además, complica las actualizaciones de la base de datos. Supóngase que se desea modificar el saldo de la cuenta C-201 de 900 a 950 €. Esta modificación debe reflejarse en las dos filas; compárese con el diseño original, en el que esto daría lugar a la actualización de una sola fila. Por tanto, las actualizaciones resultan más costosas bajo el diseño alternativo que bajo el diseño original. Cuando se lleva a cabo la actualización de la base de datos alternativa, hay que asegurarse de que *todas* las tuplas que afectan a la cuenta C-201 se actualicen, o la base de datos mostrará dos valores de saldo diferentes para la cuenta C-201.

Examíñese ahora el problema de la “imposibilidad de representar determinada información”. Supóngase que, en vez de tener las dos tablas separadas *cliente* e *impositor*, se tuviera una sola tabla, *cliente'*, que combinara la información de esas dos tablas (como puede verse en la Figura 1.5). No se puede representar directamente la información relativa a los clientes (*id_cliente*, *nombre_cliente*, *calle_cliente*, *ciudad_cliente*) a menos que el cliente tenga, como mínimo, una cuenta en el banco. Esto se debe a que las filas de *cliente'* necesitan valores de *número_cuenta*.

Una solución de este problema es introducir valores **nulos**. Los valores *nulos* indican que el valor no existe (o es desconocido). Los valores desconocidos pueden ser valores *ausentes* (el valor existe, pero no se tiene la información) o valores *desconocidos* (no se sabe si el valor existe realmente o no). Como se verá más adelante, los valores nulos resultan difíciles de tratar, y es preferible no recurrir a ellos. Si no se desea tratar con valores nulos, se puede crear un elemento concreto de información del cliente sólo si el cliente tiene cuenta en el banco (obsérvese que los clientes pueden tener un préstamo pero no tener ninguna cuenta). Además, habría que eliminar esa información cuando el cliente cerrara la cuenta. Claramente, esta situación no es deseable ya que, bajo el diseño original de la base de datos, la información de los clientes estaría disponible independientemente de si el cliente tiene cuenta en el banco o no, y sin necesidad de recurrir a los valores nulos.

1.7 Bases de datos basadas en objetos y semiestructuradas

Varias áreas de aplicaciones de los sistemas de bases de datos están limitadas por las restricciones del modelo de datos relacional. En consecuencia, los investigadores han desarrollado varios modelos de datos para tratar con estos dominios de aplicación. Los modelos de datos que se tratarán en este texto son el orientado a objetos y el relacional orientado a objetos, representativos de los modelos de datos basados en objetos, y XML, representativo de los modelos de datos semiestructurados.

1.7.1 Modelos de datos basados en objetos

El **modelo de datos orientado a objetos** se basa en el paradigma de los lenguajes de programación orientados a objetos, que actualmente se usa en gran medida. La herencia, la identidad de los objetos y la encapsulación (ocultación de la información), con métodos para ofrecer una interfaz para los objetos, están entre los conceptos principales de la programación orientada a objetos que han encontrado aplicación en el modelado de datos. El modelo de datos orientado a objetos también soporta un sistema elaborado de tipos, incluidos los tipos estructurados y las colecciones. El modelo orientado a objetos puede considerarse una extensión del modelo E-R con los conceptos de encapsulación, métodos (funciones) e identidad de los objetos.

El **modelo de datos relacional orientado a objetos** extiende el modelo relacional tradicional con gran variedad de características como los tipos estructurados y las colecciones, así como la orientación a objetos.

En el Capítulo 9 se examinan las bases de datos relacionales orientadas a objetos (es decir, las bases de datos construidas según el modelo relacional orientado a objetos), así como las bases de datos orientadas a objetos (es decir, las bases de datos construidas según el modelo de datos orientado a objetos).

1.7.2 Modelos de datos semiestructurados

Los modelos de datos semiestructurados permiten la especificación de los datos en los que cada elemento de datos del mismo tipo puede tener conjuntos de atributos diferentes. Esto los diferencia de los modelos de datos mencionados anteriormente, en los que todos los elementos de datos de un tipo dado deben tener el mismo conjunto de atributos.

El lenguaje XML se diseñó inicialmente como un modo de añadir información de marcas a los documentos de texto, pero se ha vuelto importante debido a sus aplicaciones en el intercambio de datos. XML ofrece un modo de representar los datos que tienen una estructura anidada y, además, permite una gran flexibilidad en la estructuración de los datos, lo cual es importante para ciertas clases de datos no tradicionales. En el Capítulo 10 se describe el lenguaje XML, diferentes maneras de expresar las consultas sobre datos representados en XML y la transformación de los datos XML de una forma a otra.

1.8 Almacenamiento de datos y consultas

Los sistemas de bases de datos están divididos en módulos que tratan con cada una de las responsabilidades del sistema general. Los componentes funcionales de los sistemas de bases de datos pueden dividirse grosso modo en los componentes gestor de almacenamiento y procesador de consultas.

El gestor de almacenamiento es importante porque las bases de datos suelen necesitar una gran cantidad de espacio de almacenamiento. Las bases de datos corporativas tienen un tamaño que va de los centenares de gigabytes hasta, para las bases de datos de mayor tamaño, los terabytes de datos. Un gigabyte son aproximadamente 1.000 megabytes (1.000 millones de bytes), y un terabyte es aproximadamente un millón de megabytes (1 billón de bytes). Debido a que la memoria principal de las computadoras no puede almacenar toda esta información, la información se almacena en discos. Los datos se intercambian entre los discos de almacenamiento y la memoria principal cuando sea necesario. Como el intercambio de datos con el disco es lento comparado con la velocidad de la unidad central de procesamiento, es fundamental que el sistema de base de datos estructure los datos para minimizar la necesidad de intercambio de datos entre los discos y la memoria principal.

El procesador de consultas es importante porque ayuda al sistema de bases de datos a simplificar y facilitar el acceso a los datos. Las vistas de alto nivel ayudan a conseguir este objetivo; con ellas, los usuarios del sistema no se ven molestados innecesariamente con los detalles físicos de la implementación del sistema. Sin embargo, el rápido procesamiento de las actualizaciones y de las consultas es importante. Es función del sistema de bases de datos traducir las actualizaciones y las consultas escritas en lenguajes no procedimentales, en el nivel lógico, en una secuencia eficiente de operaciones en el nivel físico.

1.8.1 Gestor de almacenamiento

Un *gestor de almacenamiento* es un módulo de programa que proporciona la interfaz entre los datos de bajo nivel almacenados en la base de datos y los programas de aplicación y las consultas remitidas al sistema. El gestor de almacenamiento es responsable de la interacción con el gestor de archivos. Los datos en bruto se almacenan en el disco mediante el sistema de archivos que suele proporcionar un sistema operativo convencional. El gestor de almacenamiento traduce las diferentes instrucciones LMD a comandos de bajo nivel del sistema de archivos. Así, el gestor de almacenamiento es responsable del almacenamiento, la recuperación y la actualización de los datos de la base de datos.

Entre los componentes del gestor de almacenamiento se encuentran:

- **Gestor de autorizaciones e integridad**, que comprueba que se satisfagan las restricciones de integridad y la autorización de los usuarios para tener acceso a los datos.

- **Gestor de transacciones**, que garantiza que la base de datos quede en un estado consistente (correcto) a pesar de los fallos del sistema, y que la ejecución concurrente de transacciones transcurra si conflictos.
- **Gestor de archivos**, que gestiona la asignación de espacio de almacenamiento de disco y las estructuras de datos usadas para representar la información almacenada en el disco.
- **Gestor de la memoria intermedia**, que es responsable de traer los datos desde el disco de almacenamiento a la memoria principal y decidir los datos a guardar en la memoria caché. El gestor de la memoria intermedia es una parte fundamental de los sistemas de bases de datos, ya que permite que la base de datos maneje tamaños de datos que son mucho mayores que el tamaño de la memoria principal.

El gestor de almacenamiento implementa varias estructuras de datos como parte de la implementación física del sistema:

- **Archivos de datos**, que almacenan la base de datos en sí misma.
- **Diccionario de datos**, que almacena metadatos acerca de la estructura de la base de datos; en particular, su esquema.
- **Índices**, que pueden proporcionar un acceso rápido a los elementos de datos. Como el índice de este libro de texto, los índices de las bases de datos facilitan punteros a los elementos de datos que tienen un valor concreto. Por ejemplo, se puede usar un índice para buscar todos los registros *cuenta* con un *número_cuenta* determinado. La asociación es una alternativa a la indexación que es más rápida en algunos casos, pero no en todos.

Se estudiarán los medios de almacenamiento, las estructuras de archivos y la gestión de la memoria intermedia en el Capítulo 11. Los métodos de acceso eficiente a los datos mediante indexación o asociación se explican en el Capítulo 12.

1.8.2 El procesador de consultas

Entre los componentes del procesador de consultas se encuentran:

- **Intérprete del LDD**, que interpreta las instrucciones del LDD y registra las definiciones en el diccionario de datos.
- **Compilador del LMD**, que traduce las instrucciones del LMD en un lenguaje de consultas a un plan de evaluación que consiste en instrucciones de bajo nivel que entienda el motor de evaluación de consultas.
Las consultas se suelen poder traducir en varios planes de ejecución alternativos, todos los cuales proporcionan el mismo resultado. El compilador del LMD también realiza **optimización de consultas**, es decir, elige el plan de evaluación de menor coste de entre todas las opciones posibles.
- **Motor de evaluación de consultas**, que ejecuta las instrucciones de bajo nivel generadas por el compilador del LMD.

La evaluación de las consultas se trata en el Capítulo 13, mientras que los métodos por los que el optimizador de consultas elige entre las estrategias de evaluación posibles se tratan en el Capítulo 14.

1.9 Gestión de transacciones

A menudo, varias operaciones sobre la base de datos forman una única unidad lógica de trabajo. Un ejemplo son las transferencia de fondos, como se vio en el Apartado 1.2, en las que se realiza un cargo en una cuenta (llámese A) y un abono en otra cuenta (llámese B). Evidentemente, resulta fundamental que, o bien tengan lugar tanto el cargo como el abono, o bien que no se produzca ninguno. Es decir, la transferencia de fondos debe tener lugar por completo o no producirse en absoluto. Este requisito

de todo o nada se denomina **atomicidad**. Además, resulta esencial que la ejecución de la transferencia de fondos preserve la consistencia de la base de datos. Es decir, el valor de la suma $A + B$ se debe preservar. Este requisito de corrección se denomina **consistencia**. Finalmente, tras la ejecución correcta de la transferencia de fondos, los nuevos valores de las cuentas A y B deben persistir, a pesar de la posibilidad de fallo del sistema. Este requisito de persistencia se denomina **durabilidad**.

Una **transacción** es un conjunto de operaciones que lleva a cabo una única función lógica en una aplicación de bases de datos. Cada transacción es una unidad de atomicidad y consistencia. Por tanto, se exige que las transacciones no violen ninguna restricción de consistencia de la base de datos. Es decir, si la base de datos era consistente cuando la transacción comenzó, debe ser consistente cuando la transacción termine con éxito. Sin embargo, durante la ejecución de una transacción, puede ser necesario permitir inconsistencias temporalmente, ya que el cargo a A o el abono a B se debe realizar en primer lugar. Esta inconsistencia temporal, aunque necesaria, puede conducir a dificultades si ocurre un fallo.

Es responsabilidad del programador definir adecuadamente las diferentes transacciones, de tal manera que cada una preserve la consistencia de la base de datos. Por ejemplo, la transacción para transferir fondos de la cuenta A a la cuenta B puede definirse como si estuviera compuesta de dos programas diferentes: uno que realiza el cargo en la cuenta A y otro que realiza el abono en la cuenta B . La ejecución de estos dos programas uno después del otro preservará realmente la consistencia. Sin embargo, cada programa en sí mismo no transforma la base de datos de un estado consistente a otro nuevo. Por tanto, estos programas no son transacciones.

Garantizar las propiedades de atomicidad y de durabilidad es responsabilidad del propio sistema de bases de datos —concretamente del **componente de gestión de transacciones**. A falta de fallos, todas las transacciones se completan con éxito y la atomicidad se consigue fácilmente. Sin embargo, debido a diversos tipos de fallos, puede que las transacciones no siempre completen su ejecución con éxito. Si se va a asegurar la propiedad de atomicidad, las transacciones fallidas no deben tener ningún efecto sobre el estado de la base de datos. Por tanto, la base de datos debe restaurarse al estado en que estaba antes de que la transacción en cuestión comience a ejecutarse. El sistema de bases de datos, por tanto, debe realizar la **recuperación de fallos**, es decir, detectar los fallos del sistema y restaurar la base de datos al estado que tenía antes de que ocurriera el fallo.

Finalmente, cuando varias transacciones actualizan la base de datos de manera concurrente, puede que no se preserve la consistencia de los datos, aunque cada una de las transacciones sea correcta. Es responsabilidad del **gestor de control de concurrencia** controlar la interacción entre las transacciones concurrentes para garantizar la consistencia de la base de datos.

Los conceptos básicos del procesamiento de transacciones se tratan en el Capítulo 15. La gestión de las transacciones concurrentes se trata en el Capítulo 16. En el Capítulo 17 se trata con detalle la recuperación de fallos.

Puede que los sistemas de bases de datos diseñados para su empleo en computadoras personales pequeños no tengan todas estas características. Por ejemplo, muchos sistemas pequeños sólo permiten que un usuario tenga acceso a la base de datos en cada momento. Otros no ofrecen copias de seguridad ni recuperación, y dejan esas tareas a los usuarios. Estas restricciones permiten un gestor de datos de menor tamaño, con menos requisitos de recursos físicos —especialmente de memoria principal. Aunque tales enfoques de bajo coste y bajas prestaciones son adecuados para bases de datos personales pequeñas, resultan inadecuados para empresas medianas y grandes.

El concepto de transacción se ha aplicado ampliamente en los sistemas y en las aplicaciones de bases de datos. Aunque el empleo inicial de las transacciones se produjo en las aplicaciones financieras, el concepto se usa ahora en aplicaciones de tiempo real de telecomunicaciones, así como en la gestión de las actividades de larga duración como el diseño de productos o los flujos de trabajo administrativos. Estas aplicaciones más amplias del concepto de transacción se estudian en el Capítulo 25.

1.10 Minería y análisis de datos

El término **minería de datos** se refiere en líneas generales al proceso de análisis semiautomático de grandes bases de datos para descubrir patrones útiles. Al igual que el descubrimiento de conocimiento en inteligencia artificial (también denominado **aprendizaje de la máquina**) o el análisis estadístico, la minería de datos intenta descubrir reglas y patrones en los datos. Sin embargo, la minería de datos

se diferencia del aprendizaje de la máquina y de la estadística en que maneja grandes volúmenes de datos, almacenados principalmente en disco. Es decir, la minería de datos trata del “descubrimiento de conocimiento en las bases de datos”.

Algunos tipos de conocimiento descubiertos en las bases de datos pueden representarse mediante un conjunto de **reglas**. Lo que sigue es un ejemplo de regla, definida informalmente: “las mujeres jóvenes con ingresos anuales superiores a 50.000 € son las personas con más probabilidades de comprar coches deportivos pequeños”. Por supuesto, esas reglas no son universalmente ciertas, sino que tienen grados de “apoyo” y de “confianza”. Otros tipos de conocimiento se representan mediante ecuaciones que relacionan diferentes variables, o mediante otros mecanismos para la predicción de los resultados cuando se conocen los valores de algunas variables.

Hay gran variedad de tipos posibles de patrones que pueden resultar útiles y se emplean diferentes técnicas para descubrir tipos de patrones diferentes. En el Capítulo 18 se estudian unos cuantos ejemplos de patrones y se ve la manera en que pueden obtenerse de las bases de datos de forma automática.

Generalmente hay un componente manual en la minería de datos, que consiste en el preprocesamiento de los datos de una manera aceptable para los algoritmos, y el postprocesamiento de los patrones descubiertos para descubrir otros nuevos que puedan resultar útiles. Puede haber también más de un tipo de patrón que pueda descubrirse en una base de datos dada, y puede ser necesaria la interacción manual para escoger los tipos de patrones útiles. Por este motivo, la minería de datos es, en realidad, un proceso semiautomático en la vida real. No obstante, en nuestra descripción se concentra la atención en el aspecto automático de la minería.

Las empresas han comenzado a explotar la creciente cantidad de datos en línea para tomar mejores decisiones sobre sus actividades, como los artículos de los que hay que tener existencias y la mejor manera de llegar a los clientes para incrementar las ventas. Muchas de sus consultas son bastante complicadas, sin embargo, y ciertos tipos de información no pueden extraerse ni siquiera usando SQL.

Se dispone de varias técnicas y herramientas para ayudar a la toma de decisiones. Varias herramientas para el análisis de datos permiten a los analistas ver los datos de diferentes maneras. Otras herramientas de análisis realizan cálculos previos de resúmenes de grandes cantidades de datos, con objeto de dar respuestas rápidas a las preguntas. El estándar SQL:1999 contiene actualmente constructores adicionales para soportar el análisis de datos.

Los datos textuales también han crecido de manera explosiva. Estos datos carecen de estructura, a diferencia de los datos rígidamente estructurados de las bases de datos relacionales. La consulta de datos textuales no estructurados se denomina *recuperación de la información*. Los sistemas de recuperación de la información tienen mucho en común con los sistemas de bases de datos —en especial, el almacenamiento y recuperación de datos en medios de almacenamiento secundarios. Sin embargo, el énfasis en el campo de los sistemas de información es diferente del de los sistemas de bases de datos, y se concentra en aspectos como las consultas basadas en palabras clave; la relevancia de los documentos para la consulta, y el análisis, clasificación e indexación de los documentos. En los Capítulos 18 y 19 se trata la ayuda a la toma de decisiones, incluyendo el procesamiento analítico en línea, la minería de datos y la recuperación de la información.

1.11 Arquitectura de las bases de datos

Ahora es posible ofrecer una visión única (Figura 1.6) de los diversos componentes de los sistemas de bases de datos y de las conexiones existentes entre ellos.

La arquitectura de los sistemas de bases de datos se ve muy influida por el sistema informático subyacente sobre el que se ejecuta el sistema de bases de datos. Los sistemas de bases de datos pueden estar centralizados o ser del tipo cliente-servidor, en los que una máquina servidora ejecuta el trabajo en nombre de multitud de máquinas clientes. Los sistemas de bases de datos pueden diseñarse también para aprovechar las arquitecturas de computadoras paralelas. Las bases de datos distribuidas se extienden por varias máquinas geográficamente separadas.

En el Capítulo 20 se trata la arquitectura general de los sistemas informáticos modernos. El Capítulo 21 describe el modo en que diversas acciones de las bases de datos, en especial el procesamiento de las consultas, pueden implementarse para aprovechar el procesamiento paralelo. El Capítulo 22 presenta varios problemas que surgen en las bases de datos distribuidas y describe el modo de afrontarlos. Entre

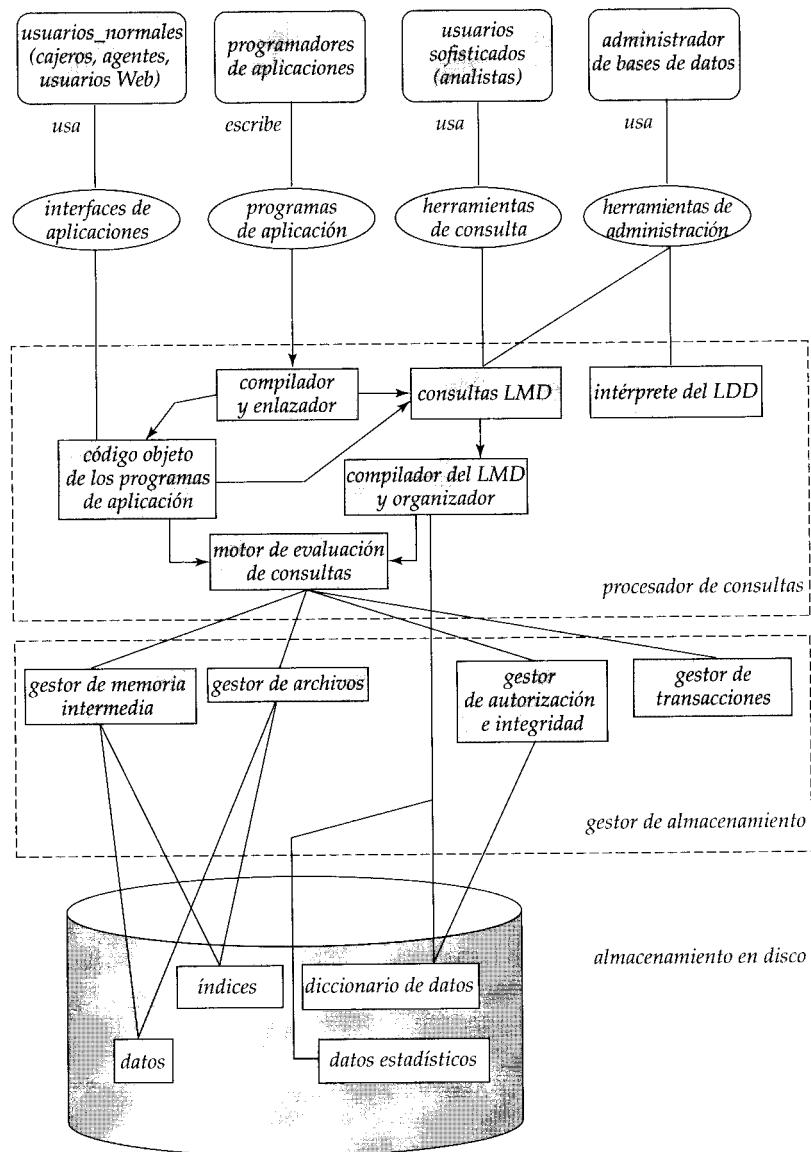


Figura 1.6 Arquitectura del sistema.

los problemas se encuentran el modo de almacenar los datos, la manera de asegurar la atomicidad de las transacciones que se ejecutan en varios sitios, cómo llevar a cabo controles de concurrencia y el modo de ofrecer alta disponibilidad en presencia de fallos. El procesamiento distribuido de las consultas y los sistemas de directorio también se describen en ese capítulo.

Hoy en día la mayor parte de los usuarios de los sistemas de bases de datos no está presente en el lugar físico en que se encuentra el sistema de bases de datos, sino que se conectan a él a través de una red. Por tanto, se puede diferenciar entre los sistemas **clientes**, en los que trabajan los usuarios remotos de la base de datos, y los sistemas **servidores**, en los que se ejecutan los sistemas de bases de datos.

Las aplicaciones de bases de datos suelen dividirse en dos o tres partes, como puede verse en la Figura 1.7. En una **arquitectura de dos capas**, la aplicación se divide en un componente que reside en la máquina cliente, que llama a la funcionalidad del sistema de bases de datos en la máquina servidora mediante instrucciones del lenguaje de consultas. Los estándares de interfaces de programas de aplicación como ODBC y JDBC se usan para la interacción entre el cliente y el servidor.

En cambio, en una **arquitectura de tres capas**, la máquina cliente actúa simplemente como una parte visible al usuario y no contiene ninguna llamada directa a la base de datos. En vez de eso, el extremo

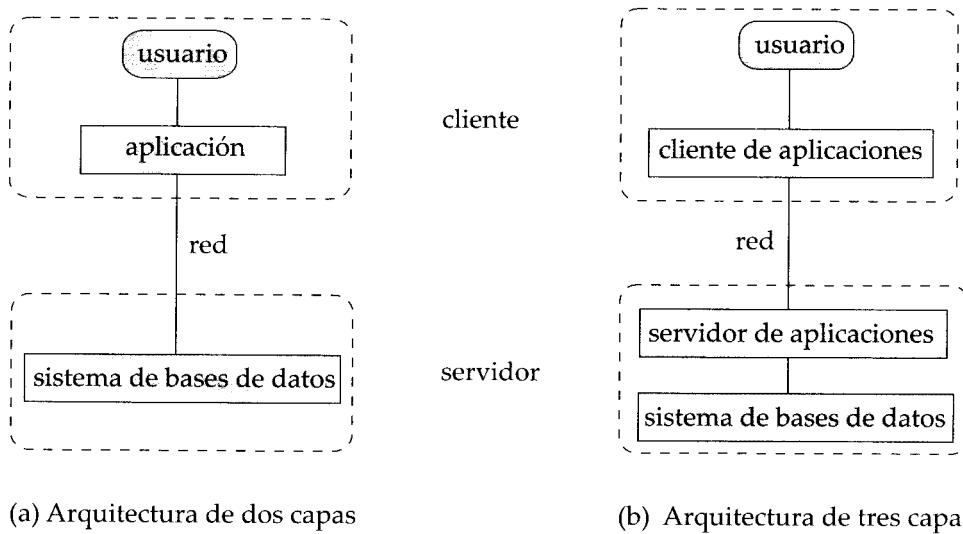


Figura 1.7 Arquitecturas de dos y tres capas.

cliente se comunica con un **servidor de aplicaciones**, generalmente mediante una interfaz de formularios. El servidor de aplicaciones, a su vez, se comunica con el sistema de bases de datos para tener acceso a los datos. La **lógica de negocio** de la aplicación, que establece las acciones que se deben realizar según las condiciones reinantes, se incorpora en el servidor de aplicaciones, en lugar de estar distribuida entre múltiples clientes. Las aplicaciones de tres capas resultan más adecuadas para aplicaciones de gran tamaño y para las aplicaciones que se ejecutan en World Wide Web.

1.12 Usuarios y administradores de bases de datos

Uno de los objetivos principales de los sistemas de bases de datos es recuperar información de la base de datos y almacenar en ella información nueva. Las personas que trabajan con una base de datos se pueden clasificar como usuarios o administradores de bases de datos.

1.12.1 Usuarios de bases de datos e interfaces de usuario

Hay cuatro tipos diferentes de usuarios de los sistemas de bases de datos, diferenciados por la forma en que esperan interactuar con el sistema. Se han diseñado diferentes tipos de interfaces de usuario para los diferentes tipos de usuarios.

- Los **usuarios normales** son usuarios no sofisticados que interactúan con el sistema invocando alguno de los programas de aplicación que se han escrito previamente. Por ejemplo, un cajero bancario que necesita transferir 50 € de la cuenta A a la cuenta B invoca un programa llamado *transferencia*. Ese programa le pide al cajero el importe de dinero que se va a transferir, la cuenta desde la que se va a transferir el dinero y la cuenta a la que se va a transferir el dinero.

Como ejemplo adicional, considérese un usuario que desea averiguar el saldo de su cuenta en World Wide Web. Ese usuario puede acceder a un formulario en el que introduce su número de cuenta. Un programa de aplicación en el servidor Web recupera entonces el saldo de la cuenta, usando el número de cuenta proporcionado, y devuelve la información al usuario.

La interfaz de usuario habitual para los usuarios normales es una interfaz de formularios, donde el usuario puede llenar los campos correspondientes del formulario. Los usuarios normales también pueden limitarse a leer *informes* generados por la base de datos.

- Los **programadores de aplicaciones** son profesionales informáticos que escriben programas de aplicación. Los programadores de aplicaciones pueden elegir entre muchas herramientas para desarrollar las interfaces de usuario. Las herramientas de **desarrollo rápido de aplicaciones**

(DRA) son herramientas que permiten al programador de aplicaciones crear formularios e informes con un mínimo esfuerzo de programación.

- Los **usuarios sofisticados** interactúan con el sistema sin escribir programas. En su lugar, formulan sus consultas en un lenguaje de consultas de bases de datos. Remiten cada una de las consultas al **procesador de consultas**, cuya función es dividir las instrucciones LMD en instrucciones que el gestor de almacenamiento entienda. Los analistas que remiten las consultas para explorar los datos de la base de datos entran en esta categoría.
- Los **usuarios especializados** son usuarios sofisticados que escriben aplicaciones de bases de datos especializadas que no encajan en el marco tradicional del procesamiento de datos. Entre estas aplicaciones están los sistemas de diseño asistido por computadora, los sistemas de bases de conocimientos y los sistemas expertos, los sistemas que almacenan datos con tipos de datos complejos (por ejemplo, los datos gráficos y los datos de sonido) y los sistemas de modelado del entorno. En el Capítulo 9 se estudian varias de estas aplicaciones.

1.12.2 Administrador de bases de datos

Una de las principales razones de usar SGBDs es tener un control centralizado tanto de los datos como de los programas que tienen acceso a esos datos. La persona que tiene ese control central sobre el sistema se denomina **administrador de bases de datos (ABD)**. Las funciones del ABD incluyen:

- La **definición del esquema**. El ABD crea el esquema original de la base de datos mediante la ejecución de un conjunto de instrucciones de definición de datos en el LDD.
- La **definición de la estructura y del método de acceso**.
- La **modificación del esquema y de la organización física**. El ABD realiza modificaciones en el esquema y en la organización física para reflejar las necesidades cambiantes de la organización, o para alterar la organización física a fin de mejorar el rendimiento.
- La **concesión de autorización para el acceso a los datos**. Mediante la concesión de diferentes tipos de autorización, el administrador de bases de datos puede regular las partes de la base de datos a las que puede tener acceso cada usuario. La información de autorización se guarda en una estructura especial del sistema que el SGBD consulta siempre que alguien intenta tener acceso a los datos del sistema.
- El **mantenimiento rutinario**. Algunos ejemplos de las actividades de mantenimiento rutinario del administrador de la base de datos son:
 - Copia de seguridad periódica de la base de datos, bien sobre cinta o sobre servidores remotos, para impedir la pérdida de datos en caso de desastres como las inundaciones.
 - Asegurarse de que se dispone de suficiente espacio libre en disco para las operaciones normales y aumentar el espacio en disco según sea necesario.
 - Supervisar los trabajos que se ejecuten en la base de datos y asegurarse de que el rendimiento no se degrade debido a que algún usuario haya remitido tareas muy costosas.

1.13 Historia de los sistemas de bases de datos

El procesamiento de datos impulsa el crecimiento de las computadoras, como lo ha hecho desde los primeros días de las computadoras comerciales. De hecho, la automatización de las tareas de procesamiento de datos precede a las computadoras. Las tarjetas perforadas, inventadas por Herman Hollerith, se emplearon a principios del siglo XX para registrar los datos del censo de Estados Unidos, y se usaron sistemas mecánicos para procesar las tarjetas y para tabular los resultados. Las tarjetas perforadas se usaron posteriormente con profusión como medio para introducir datos en las computadoras.

Las técnicas de almacenamiento y de procesamiento de datos han evolucionado a lo largo de los años:

- **Años cincuenta y primeros años sesenta:** se desarrollaron las cintas magnéticas para el almacenamiento de datos. Las tareas de procesamiento de datos como la elaboración de nóminas se

automatizaron, con los datos almacenados en cintas. El procesamiento de datos consistía en leer datos de una o varias cintas y escribir datos en una nueva cinta. Los datos también se podían introducir desde paquetes de tarjetas perforadas e imprimirse en impresoras. Por ejemplo, los aumentos de sueldo se procesaban introduciendo los aumentos en las tarjetas perforadas y leyendo el paquete de cintas perforadas de manera sincronizada con una cinta que contenía los detalles principales de los salarios. Los registros debían estar en el mismo orden. Los aumentos de sueldo se añadían a los sueldos leídos de la cinta maestra y se escribían en una nueva cinta; esa nueva cinta se convertía en la nueva cinta maestra.

Las cintas (y los paquetes de tarjetas perforadas) sólo se podían leer secuencialmente, y el tamaño de datos era mucho mayor que la memoria principal; por tanto, los programas de procesamiento de datos se veían obligados a procesar los datos en un orden determinado, leyendo y mezclando datos de las cintas y de los paquetes de tarjetas perforadas.

- **Finales de los años sesenta y años setenta:** el empleo generalizado de los discos duros a finales de los años sesenta modificó en gran medida la situación del procesamiento de datos, ya que permitieron el acceso directo a los datos. La ubicación de los datos en disco no era importante, ya que se podía tener acceso a cualquier posición del disco en sólo unas decenas de milisegundos. Los datos se liberaron así de la tiranía de la secuencialidad. Con los discos pudieron crearse las bases de datos de red y las bases de datos jerárquicas, que permitieron que las estructuras de datos como las listas y los árboles pudieran almacenarse en disco. Los programadores pudieron crear y manipular estas estructuras de datos.

El artículo histórico de Codd [1970] definió el modelo relacional y las formas no procedimentales de consultar los datos en el modelo relacional, y así nacieron las bases de datos relacionales. La simplicidad del modelo relacional y la posibilidad de ocultar completamente los detalles de implementación a los programadores resultaron realmente atractivas. Codd obtuvo posteriormente el prestigioso premio Turing de la ACM (Association of Computing Machinery, asociación de maquinaria informática) por su trabajo.

- **Años ochenta:** aunque académicamente interesante, el modelo relacional no se usó inicialmente en la práctica debido a sus inconvenientes en cuanto a rendimiento; las bases de datos relacionales no podían igualar el rendimiento de las bases de datos de red y jerárquicas existentes. Esta situación cambió con System R, un proyecto innovador del centro de investigación IBM Research que desarrolló técnicas para la construcción de un sistema de bases de datos relacionales eficiente. En Astrahan et al. [1976] y Chamberlin et al. [1981] se pueden encontrar excelentes visiones generales de System R. El prototipo de System R completamente funcional condujo al primer producto de bases de datos relacionales de IBM: SQL/DS. Los primeros sistemas comerciales de bases de datos relacionales, como DB2 de IBM, Oracle, Ingres y Rdb de DEC, desempeñaron un importante papel en el desarrollo de técnicas para el procesamiento eficiente de las consultas declarativas. En los primeros años ochenta las bases de datos relacionales habían llegado a ser competitivas frente a los sistemas de bases de datos jerárquicas y de red incluso en cuanto a rendimiento. Las bases de datos relacionales eran tan sencillas de usar que finalmente reemplazaron a las bases de datos jerárquicas y de red; los programadores que usaban esas bases de datos se veían obligados a tratar muchos detalles de implementación de bajo nivel y tenían que codificar sus consultas de forma procedural. Lo que era aún más importante, tenían que tener presente el rendimiento durante el diseño de los programas, lo que suponía un gran esfuerzo. En cambio, en las bases de datos relacionales, casi todas estas tareas de bajo nivel las realiza de manera automática el sistema de bases de datos, lo que libera al programador para que se centre en el nivel lógico. Desde su obtención de liderazgo en los años ochenta, el modelo relacional ha reinado sin discusión entre todos los modelos de datos.

Los años ochenta también fueron testigos de una gran investigación en las bases de datos paralelas y distribuidas, así como del trabajo inicial en las bases de datos orientadas a objetos.

- **Primeros años noventa:** el lenguaje SQL se diseñó fundamentalmente para las aplicaciones de ayuda a la toma de decisiones, que son intensivas en consultas, mientras que el objetivo principal de las bases de datos en los años ochenta eran las aplicaciones de procesamiento de trans-

sacciones, que son intensivas en actualizaciones. La ayuda a la toma de decisiones y las consultas volvieron aemerger como una importante área de aplicación para las bases de datos. El uso de las herramientas para analizar grandes cantidades de datos experimentó un gran crecimiento.

En esta época muchas marcas de bases de datos introdujeron productos de bases de datos paralelas. Las diferentes marcas de bases de datos también comenzaron a añadir soporte relacional orientado a objetos a sus bases de datos.

- **Finales de los años noventa:** el principal acontecimiento fue el crecimiento explosivo de World Wide Web. Las bases de datos se implantaron mucho más ampliamente que nunca. Los sistemas de bases de datos tenían que soportar tasas de procesamiento de transacciones muy elevadas, así como una fiabilidad muy alta y tener disponibilidad 24×7 (disponibilidad 24 horas al día y 7 días a la semana, lo que significa que no hay momentos de inactividad debidos a actividades de mantenimiento planificadas). Los sistemas de bases de datos también tenían que soportar interfaces Web para los datos.
- **Principios del siglo XXI:** los principios del siglo XXI han sido testigos de la emergencia de XML y de su lenguaje de consultas asociado, XQuery, como nueva tecnología de las bases de datos. Todavía es pronto para decir el papel que XML desempeñará en las bases de datos futuras. En este periodo también se ha podido presenciar el crecimiento de las técnicas de “informática autónoma/administración automática” para la minimización del esfuerzo de administración.

1.14 Resumen

- Un **sistema gestor de bases de datos (SGBD)** consiste en un conjunto de datos interrelacionados y en un conjunto de programas para tener acceso a esos datos. Los datos describen una empresa concreta.
- El objetivo principal de un SGBD es proporcionar un entorno que sea tanto conveniente como eficiente para las personas que lo usan para la recuperación y almacenamiento de información.
- Los sistemas de bases de datos resultan ubicuos hoy en día, y la mayor parte de la gente interactúa, directa o indirectamente, con bases de datos muchas veces al día.
- Los sistemas de bases de datos se diseñan para almacenar grandes cantidades de información. La gestión de los datos implica tanto la definición de estructuras para el almacenamiento de la información como la provisión de mecanismos para la manipulación de la información. Además, los sistemas de bases de datos deben preocuparse de la seguridad de la información almacenada, en caso de caídas del sistema o de intentos de acceso sin autorización. Si los datos deben compartirse entre varios usuarios, el sistema debe evitar posibles resultados anómalos.
- Uno de los propósitos principales de los sistemas de bases de datos es ofrecer a los usuarios una visión abstracta de los datos. Es decir, el sistema oculta ciertos detalles de la manera en que los datos se almacenan y mantienen.
- Por debajo de la estructura de la base de datos se halla el **modelo de datos**: un conjunto de herramientas conceptuales para describir los datos, las relaciones entre los datos, la semántica de los datos y las restricciones de los datos.
- Un **lenguaje de manipulación de datos (LMD)** es un lenguaje que permite a los usuarios tener acceso a los datos o manipularlos. Los LMDs no procedimentales, que sólo necesitan que el usuario especifique los datos que necesita, sin especificar exactamente la manera de obtenerlos, se usan mucho hoy en día.
- Un **lenguaje de definición de datos (LDD)** es un lenguaje para la especificación del esquema de la base de datos y otras propiedades de los datos.
- El modelo de datos relacional es el más implantado para el almacenamiento de datos en las bases de datos. Otros modelos de datos son el modelo de datos orientado a objetos, el modelo relacional orientado a objetos y los modelos de datos semiestructurados.

- El diseño de bases de datos supone sobre todo el diseño del esquema de la base de datos. El modelo de datos entidad-relación (E-R) es un modelo de datos muy usado para el diseño de bases de datos. Proporciona una representación gráfica conveniente para ver los datos, las relaciones y las restricciones.
- Cada sistema de bases de datos tiene varios subsistemas:
 - El subsistema **gestor de almacenamiento** proporciona la interfaz entre los datos de bajo nivel almacenados en la base de datos y los programas de aplicación y las consultas remitidas al sistema.
 - El subsistema **procesador de consultas** compila y ejecuta instrucciones LDD y LMD.
- El **gestor de transacciones** garantiza que la base de datos permanezca en un estado consistente (correcto) a pesar de los fallos del sistema. El gestor de transacciones garantiza que la ejecución de las transacciones concurrentes se produzca sin conflictos.
- Las aplicaciones de bases de datos suelen dividirse en una fachada que se ejecuta en las máquinas clientes y una parte que se ejecuta en segundo plano. En las arquitecturas de dos capas la fachada se comunica directamente con una base de datos que se ejecuta en segundo plano. En las arquitecturas de tres capas la parte en segundo plano se divide a su vez en un servidor de aplicaciones y un servidor de bases de datos.
- Los usuarios de bases de datos se pueden dividir en varias clases, y cada clase de usuario suele usar un tipo diferente de interfaz para la base de datos.

Términos de repaso

- Sistema gestor de bases de datos (SGBD).
- Aplicaciones de sistemas de bases de datos.
- Sistemas de archivos.
- Inconsistencia de datos.
- Restricciones de consistencia.
- Vistas de datos.
- Abstracción de datos.
- Ejemplar de la base de datos.
- Esquema.
 - Esquema de la base de datos.
 - Esquema físico.
 - Esquema lógico.
- Independencia física de los datos.
- Modelos de datos.
 - Modelo entidad-relación.
- Modelo de datos relacional.
- Modelo de datos orientado a objetos.
- Modelo de datos relacional orientado a objetos.
- Lenguajes de bases de datos.
 - Lenguaje de definición de datos.
 - Lenguaje de manipulación de datos.
 - Lenguaje de consultas.
- Diccionario de datos.
- Metadatos.
- Transacciones.
- Conurrencia.
- Programa de aplicación.
- Administrador de bases de datos (ABD).
- Máquinas cliente y servidor.

Ejercicios prácticos

- 1.1 En este capítulo se han descrito varias ventajas importantes de los sistemas gestores de bases de datos. ¿Cuáles son sus dos inconvenientes?
- 1.2 Indíquense siete lenguajes de programación que sean procedimentales y dos que no lo sean. ¿Qué grupo es más fácil de aprender a usar? Explíquese la respuesta.
- 1.3 Indíquense seis pasos importantes que se deben dar para configurar una base de datos para una empresa dada.

- 1.4 Considérese un *array* de enteros bidimensional de tamaño $n \times m$ que se va a usar en el lenguaje de programación preferido del lector. Usando el *array* como ejemplo, ilústrese la diferencia (a) entre los tres niveles de abstracción de datos y (b) entre el esquema y los ejemplares.

Ejercicios

- 1.5 Indíquense cuatro aplicaciones que se hayan usado que sea muy posible que utilicen un sistema de bases de datos para almacenar datos persistentes.
- 1.6 Indíquense cuatro diferencias significativas entre un sistema de procesamiento de archivos y un SGBD.
- 1.7 Explíquese la diferencia entre independencia de datos física y lógica.
- 1.8 Indíquense cinco responsabilidades del sistema gestor de bases de datos. Para cada responsabilidad, explíquense los problemas que surgirían si no se asumiera esa responsabilidad.
- 1.9 Indíquense al menos dos razones para que los sistemas de bases de datos soporten la manipulación de datos mediante un lenguaje de consultas declarativo como SQL, en vez de limitarse a ofrecer una biblioteca de funciones de C o de C++ para llevar a cabo la manipulación de los datos.
- 1.10 Explíquense los problemas que causa el diseño de la tabla de la Figura 1.5.
- 1.11 ¿Cuáles son las cinco funciones principales del administrador de bases de datos?

Notas bibliográficas

A continuación se ofrece una relación de libros de propósito general, colecciones de artículos de investigación y sitios Web sobre bases de datos. Los capítulos siguientes ofrecen referencias a material sobre cada tema descrito en ese capítulo.

Codd [1970] es el artículo histórico que introdujo el modelo relacional.

Entre los libros de texto que tratan los sistemas de bases de datos están Abiteboul et al. [1995], Date [2003], Elmasri y Navathe [2003], O'Neil y O'Neil [2000], Ramakrishnan y Gehrke [2002], Garcia-Molina et al. [2001] y Ullman [1988]. El tratamiento del procesamiento de transacciones en libros de texto se puede encontrar en Bernstein y Newcomer [1997] y Gray y Reuter [1993].

Varios libros incluyen colecciones de artículos de investigación sobre la gestión de las bases de datos. Entre éstos se encuentran Bancilhon y Buneman [1990], Date [1986], Date [1990], Kim [1995], Zaniolo et al. [1997] y Hellerstein y Stonebraker [2005].

Un repaso de los logros en la gestión de bases de datos y una valoración de los desafíos en la investigación futura aparece en Silberschatz et al. [1990], Silberschatz et al. [1996], Bernstein et al. [1998] y Abiteboul et al. [2003]. La página inicial del grupo de interés especial de la ACM en gestión de datos (www.acm.org/sigmod) ofrece gran cantidad de información sobre la investigación en bases de datos. Los sitios Web de los fabricantes de bases de datos (véase a continuación el apartado *Herramientas*) proporciona detalles acerca de sus respectivos productos.

Herramientas

Hay gran número de sistemas de bases de datos comerciales actualmente en uso. Entre los principales están: DB2 de IBM (www.IBM.com/software/data), Oracle (www.oracle.com), SQL Server de Microsoft (www.microsoft.com/SQL), Informix (www.informix.com) (ahora propiedad de IBM) y Sybase (www.sybase.com). Algunos de estos sistemas están disponibles gratuitamente para uso personal o no comercial, o para desarrollo, pero no para su implantación real.

También hay una serie de sistemas de bases de datos gratuitos o de dominio público; los más usados son MySQL (www.mySQL.com) y PostgreSQL (www.postgreSQL.org).

Una lista más completa de enlaces a sitios Web de fabricantes y a otras informaciones se encuentra disponible en la página inicial de este libro, en <http://www.mhe.es/universidad/informatica/fundamentos>.

Bases de datos relacionales

Un modelo de datos es un conjunto de herramientas conceptuales para la descripción de los datos, las relaciones entre ellos, su semántica y las restricciones de consistencia. Esta parte centra la atención en el modelo relacional.

El modelo relacional utiliza un conjunto de tablas para representar tanto los datos como las relaciones entre ellos. Su simplicidad conceptual ha conducido a su adopción generalizada; actualmente, una amplia mayoría de los productos de bases de datos se basan en el modelo relacional. En la Parte 9 se expone una visión general de cuatro sistemas de bases de datos relacionales muy utilizados.

Aunque el modelo relacional, que se trata en el Capítulo 2, describe los datos en los niveles lógico y de vistas, es un modelo de datos de nivel inferior comparado con el modelo entidad-relación, que se explica en la Parte 2.

Para poner a disposición de los usuarios los datos de una base de datos relacional hay que abordar varios aspectos. Uno de ellos es determinar uno de los distintos lenguajes de consultas para expresar las peticiones de datos. Los Capítulos 3 y 4 tratan el lenguaje SQL, que es el lenguaje de consultas más utilizado hoy en día. El Capítulo 5 trata en primer lugar dos lenguajes de consultas formales, el cálculo relacional de tuplas y el cálculo relacional de dominios, que son lenguajes declarativos basados en la lógica matemática. Estos dos lenguajes formales constituyen la base para dos lenguajes más amigables para los usuarios: QBE y Datalog, que se estudian en ese capítulo.

Otro aspecto es la integridad y la protección de los datos; las bases de datos deben proteger sus datos del daño que puedan causar las acciones de los usuarios, sean involuntarias o intencionadas. El componente de mantenimiento de la integridad de la base de datos garantiza que las actualizaciones no violen las restricciones de integridad que se hayan especificado para los datos. El componente de protección de la base de datos incluye el control de acceso para restringir las acciones permitidas a cada usuario. El Capítulo 4 trata los problemas de integridad y de protección. Los problemas de protección y de integridad están presentes independientemente del modelo de datos pero, en aras de la concisión, se estudiarán en el contexto del modelo relacional.

El modelo relacional

El modelo relacional es hoy en día el principal modelo de datos para las aplicaciones comerciales de procesamiento de datos. Ha conseguido esa posición destacada debido a su simplicidad, lo cual facilita el trabajo del programador en comparación con modelos anteriores, como el de red y el jerárquico.

En este capítulo se estudian en primer lugar los fundamentos del modelo relacional. A continuación se describe el álgebra relacional, que se usa para especificar las solicitudes de información. El álgebra relacional no es cómoda de usar, pero sirve de base formal para lenguajes de consultas que sí lo son y que se estudiarán más adelante, incluido el ampliamente usado lenguaje de consultas SQL, el cual se trata con detalle en los Capítulos 3 y 4.

Existe una amplia base teórica para las bases de datos relacionales. En este capítulo se estudia la parte de esa base teórica referida a las consultas. En los Capítulos 6 y 7 se examinarán aspectos de la teoría de las bases de datos relacionales que ayudan en el diseño de esquemas de bases de datos relacionales, mientras que en los Capítulos 13 y 14 se estudian aspectos de la teoría que se refieren al procesamiento eficiente de consultas.

2.1 La estructura de las bases de datos relacionales

Una base de datos relacional consiste en un conjunto de **tablas**, a cada una de las cuales se le asigna un nombre exclusivo. Cada fila de la tabla representa una *relación* entre un conjunto de valores. De manera informal, cada tabla es un conjunto de entidades, y cada fila es una entidad, tal y como se estudió en el Capítulo 1. Dado que cada tabla es un conjunto de tales relaciones, hay una fuerte correspondencia entre el concepto de *tabla* y el concepto matemático de *relación*, del que toma su nombre el modelo de datos relacional. A continuación se introduce el concepto de relación.

En este capítulo se usarán varias relaciones diferentes para ilustrar los diversos conceptos subyacentes al modelo de datos relacional. Estas relaciones representan parte de una entidad bancaria. Puede que no se correspondan con el modo en que se pueda estructurar realmente una base de datos bancaria, pero así se simplificará la presentación. Los criterios sobre la adecuación de las estructuras relacionales se estudian con gran detalle en los Capítulos 6 y 7.

2.1.1 Estructura básica

Considérese la tabla *cuenta* de la Figura 2.1. Tiene tres cabeceras de columna: *número_cuenta*, *nombre_sucursal* y *saldo*. Siguiendo la terminología del modelo relacional, se puede hacer referencia a estas cabeceras como **atributos**. Para cada atributo hay un conjunto de valores permitidos, denominado **dominio** de ese atributo. Para el atributo *nombre_sucursal*, por ejemplo, el dominio es el conjunto de todos los nombres de sucursal. Supóngase que D_1 denota el conjunto de todos los números de cuenta, D_2 el conjunto de todos los nombres de sucursal y D_3 el conjunto de todos los saldos. Todas las filas de *cuenta* deben consistir en una tupla (v_1, v_2, v_3) , donde v_1 es un número de cuenta (es decir, v_1 está en

número_cuenta	nombre_sucursal	saldo
C-101	Centro	500
C-102	Navacerrada	400
C-201	Galapagar	900
C-215	Becerril	700
C-217	Galapagar	750
C-222	Moralzarzal	700
C-305	Collado Mediano	350

Figura 2.1 La relación *cuenta*.

el dominio D_1), v_2 es un nombre de sucursal (v_2 en D_2) y v_3 es un saldo (v_3 en D_3). En general, *cuenta* sólo contendrá un subconjunto del conjunto de todas las filas posibles. Por tanto, *cuenta* será un subconjunto de

$$D_1 \times D_2 \times D_3$$

En general, una **tabla** de n atributos debe ser un subconjunto de

$$D_1 \times D_2 \times \cdots \times D_{n-1} \times D_n$$

Los matemáticos definen las **relaciones** como subconjuntos del producto cartesiano de la lista de dominios. Esta definición se corresponde de manera casi exacta con la definición de *tabla* dada anteriormente. La única diferencia es que aquí se han asignado nombres a los atributos, mientras que los matemáticos sólo usan “nombres” numéricos, usando el entero 1 para denotar el atributo cuyo dominio aparece en primer lugar de la lista de dominios, 2 para el atributo cuyo dominio aparece en segundo lugar, etc. Como las tablas son, esencialmente, relaciones, se usarán los términos matemáticos **relación** y **tupla** en lugar de los términos **tabla** y **fila**. Una **variable tupla** es una variable que representa una tupla; en otras palabras, una variable tupla es una variable cuyo dominio es el conjunto de todas las tuplas.

En la relación *cuenta* de la Figura 2.1 hay siete tuplas. Supóngase que la variable tupla t hace referencia a la primera tupla de la relación. Se usa la notación $t[número_cuenta]$ para denotar el valor de t en el atributo *número_cuenta*. Por tanto, $t[número_cuenta] = "C-101"$ y $t[nombre_sucursal] = "Centro"$. De manera alternativa, se puede escribir $t[1]$ para denotar el valor de la tupla t en el primer atributo (*número_cuenta*), $t[2]$ para denotar *nombre_sucursal*, etc. Dado que las relaciones son conjuntos de tuplas, se usa la notación matemática $t \in r$ para denotar que la tupla t está en la relación r .

El orden en que aparecen las tuplas en cada relación es irrelevante, dado que una relación es un *conjunto* de tuplas. Por tanto, no importa si las tuplas de una relación aparecen ordenadas, como en la Figura 2.1, o desordenadas, como en la Figura 2.2; las relaciones de las dos figuras son la misma, ya que las dos contienen el mismo conjunto de tuplas.

Se exige que, para todas las relaciones r , los dominios de todos los atributos de r sean atómicos. Un dominio es **atómico** si los elementos del dominio se consideran unidades indivisibles. Por ejemplo, el conjunto de los enteros es un dominio atómico, pero el conjunto de todos los conjuntos de enteros

número_cuenta	nombre_sucursal	saldo
C-101	Centro	500
C-215	Becerril	700
C-102	Navacerrada	400
C-305	Collado Mediano	350
C-201	Galapagar	900
C-222	Moralzarzal	700
C-217	Galapagar	750

Figura 2.2 La relación *cuenta* con las tuplas desordenadas.

es un dominio no atómico. La diferencia es que no se suele considerar que los enteros tengan partes constituyentes, pero sí se considera que los conjuntos de enteros las tienen; por ejemplo, los enteros que forman cada conjunto. Lo importante no es lo que sea el propio dominio, sino la manera en que se usan los elementos del dominio en la base de datos. El dominio de todos los enteros sería no atómico si se considerara que cada entero es una lista ordenada de cifras. En todos los ejemplos se supondrá que los dominios son atómicos. En el Capítulo 9 se estudiarán extensiones al modelo de datos relacional para permitir dominios no atómicos.

Es posible que varios atributos tengan el mismo dominio. Por ejemplo, supóngase la relación *cliente* con los tres atributos *nombre_cliente*, *calle_cliente* y *ciudad_cliente* y una relación *empleado* con el atributo *nombre_empleado*. Es posible que los atributos *nombre_cliente* y *nombre_empleado* tengan el mismo dominio, el conjunto de todos los nombres de persona, que en el nivel físico es el conjunto de todas las cadenas de caracteres. Los dominios de *saldo* y *nombre_sucursal*, por otra parte, deberían ser distintos. Quizás sea menos evidente si *nombre_cliente* y *nombre_sucursal* deberían tener el mismo dominio. En el nivel físico, tanto los nombres de los clientes como los nombres de las sucursales son cadenas de caracteres. Sin embargo, en el nivel lógico puede que se desee que *nombre_cliente* y *nombre_sucursal* tengan dominios diferentes.

Un valor de dominio que es miembro de todos los dominios posibles es el valor **nulo**, que indica que el valor es desconocido o no existe. Por ejemplo, supóngase que se incluye el atributo *número_teléfono* en la relación *cliente*. Puede ocurrir que algún cliente no tenga número de teléfono, o que su número de teléfono no figure en la guía. Entonces habrá que recurrir a los valores nulos para indicar que el valor es desconocido o que no existe. Más adelante se verá que los valores nulos crean algunas dificultades cuando se tiene acceso a la base de datos o se la actualiza y que, por tanto, deben eliminarse si es posible. Se supondrá inicialmente que no hay valores nulos y en el Apartado 2.5 se describirá el efecto de los valores nulos en las diferentes operaciones.

2.1.2 Esquema de la base de datos

Cuando se habla de bases de datos se debe diferenciar entre el **esquema de la base de datos**, que es el diseño lógico de la misma, y el **ejemplar de la base de datos**, que es una instantánea de los datos de la misma en un momento dado.

El concepto de relación se corresponde con el concepto de variable de los lenguajes de programación. El concepto de **esquema de la relación** se corresponde con el concepto de definición de tipos de los lenguajes de programación.

Resulta conveniente dar nombre a los esquemas de las relaciones, igual que se dan nombres a las definiciones de los tipos en los lenguajes de programación. Se adopta el convenio de usar nombres en minúsculas para las relaciones y nombres que comiencen por una letra mayúscula para los esquemas de las relaciones. Siguiendo esta notación se usará *Esquema_cuenta* para denotar el esquema de la relación *cuenta*. Por tanto,

$$\text{Esquema_cuenta} = (\text{número_cuenta}, \text{nombre_sucursal}, \text{saldo})$$

Se denota el hecho de que *cuenta* es una relación de *Esquema_cuenta* mediante

$$\text{cuenta}(\text{Esquema_cuenta})$$

En general, los esquemas de las relaciones consisten en una lista de los atributos y de sus dominios correspondientes. La definición exacta del dominio de cada atributo no será relevante hasta que se estude el lenguaje SQL en los Capítulos 3 y 4.

El concepto de **ejemplar de la relación** se corresponde con el concepto de valor de una variable en los lenguajes de programación. El valor de una variable dada puede cambiar con el tiempo; de manera parecida, el contenido del ejemplar de una relación puede cambiar con el tiempo cuando la relación se actualiza. Sin embargo, se suele decir simplemente “relación” cuando realmente se quiere decir “ejemplar de la relación”.

Como ejemplo de ejemplar de una relación, considérese la relación *sucursal* de la Figura 2.3. El esquema de esa relación es

<i>nombre_sucursal</i>	<i>ciudad_sucursal</i>	<i>activos</i>
Becerril	Aluche	400.000
Centro	Arganzuela	9.000.000
Collado Mediano	Aluche	8.000.000
Galapagar	Arganzuela	7.100.000
Moralzarzal	La Granja	2.100.000
Navacerrada	Aluche	1.700.000
Navas de la Asunción	Alcalá de Henares	300.000
Segovia	Cerceda	3.700.000

Figura 2.3 La relación *sucursal*.

$$\text{Esquema_sucursal} = (\text{nombre_sucursal}, \text{ciudad_sucursal}, \text{activos})$$

Obsérvese que el atributo *nombre_sucursal* aparece tanto en *Esquema_sucursal* como en *Esquema_cuenta*. Esta duplicidad no es una coincidencia. Más bien, usar atributos comunes en los esquemas de las relaciones es una manera de relacionar las tuplas de relaciones diferentes. Por ejemplo, supóngase que se desea obtener información sobre todas las cuentas abiertas en sucursales ubicadas en Arganzuela. Primero se busca en la relación *sucursal* para encontrar los nombres de todas las sucursales situadas en Arganzuela. A continuación y para cada una de ellas, se examina la relación *cuenta* para encontrar la información sobre las cuentas abiertas en esa sucursal.

Siguiendo con el ejemplo bancario, se necesita una relación que describa información sobre los clientes. El esquema de la relación es:

$$\text{Esquema_cliente} = (\text{nombre_cliente}, \text{calle_cliente}, \text{ciudad_cliente})$$

La Figura 2.4 muestra un ejemplo de la relación *cliente* (*Esquema_cliente*). Obsérvese que se ha omitido el atributo *id_cliente*, que se usó en el Capítulo 1, ya que se considerarán esquemas de relación más pequeños en nuestro ejemplo de una base de datos bancaria. Se da por supuesto que el nombre de cliente identifica únicamente a cada cliente—obviamente, puede que esto no sea cierto en el mundo real, pero la suposición hace los ejemplos más sencillos de entender. En una base de datos del mundo real, *id_cliente* (que podría ser *número_seguridad_social* o un identificador generado por el banco) serviría para identificar únicamente a los clientes.

También se necesita una relación que describa la asociación entre los clientes y las cuentas. El esquema de la relación que describe esta asociación es:

$$\text{Esquema_impositor} = (\text{nombre_cliente}, \text{número_cuenta})$$

<i>nombre_cliente</i>	<i>calle_cliente</i>	<i>ciudad_cliente</i>
Abril	Preciados	Valsaín
Amo	Embajadores	Arganzuela
Badorrey	Delicias	Valsaín
Fernández	Jazmín	León
Gómez	Carretas	Cerceda
González	Arenal	La Granja
López	Mayor	Peguerinos
Pérez	Carretas	Cerceda
Rodríguez	Yeserías	Cádiz
Rupérez	Ramblas	León
Santos	Mayor	Peguerinos
Valdivieso	Goya	Vigo

Figura 2.4 La relación *cliente*.

nombre_cliente	número_cuenta
Abril	C-305
Gómez	C-215
González	C-101
González	C-201
López	C-102
Rupérez	C-222
Santos	C-217

Figura 2.5 La relación *impositor*.

La Figura 2.5 muestra un ejemplo de la relación *impositor* (*Esquema_impositor*).

Puede parecer que para este ejemplo bancario se podría tener sólo un esquema de relación, en vez de tener varios. Es decir, puede resultar más sencillo para el usuario pensar en términos de un único esquema de relación, en lugar de en varios esquemas. Supóngase que sólo se usara una relación para el ejemplo, con el esquema

$$(nombre_sucursal, ciudad_sucursal, activos, nombre_cliente, calle_cliente \\ ciudad_cliente, número_cuenta, saldo)$$

Obsérvese que, si un cliente tiene varias cuentas, hay que repetir su dirección una vez por cada cuenta. Es decir, hay que repetir varias veces parte de la información. Esta repetición malgasta espacio, pero se evita mediante el empleo de varias relaciones mostradas anteriormente.

Además, si una sucursal no tiene ninguna cuenta (por ejemplo, una sucursal recién abierta que todavía no tenga clientes), no se puede construir una tupla completa en la relación única anterior, dado que no hay todavía ningún dato disponible referente a *cliente* ni a *cuenta*. Para representar las tuplas incompletas hay que usar valores *nulos* que indiquen que ese valor es desconocido o no existe. Por tanto, en el ejemplo presente, los valores de *nombre_cliente*, *calle_cliente*, etc., deben ser nulos. Al emplear varias relaciones se puede representar la información de las sucursales del banco sin clientes sin necesidad de valores nulos. Se usa simplemente una tupla en *Esquema_sucursal* para representar la información de la sucursal, y sólo se crean tuplas en los otros esquemas cuando esté disponible la información correspondiente.

En el Capítulo 7 se estudiarán los criterios para decidir cuándo un conjunto de esquemas de relaciones es más adecuado que otro en términos de repetición de la información y de la existencia de valores nulos. Por ahora se supondrá que los esquemas de las relaciones vienen dados de antemano.

Se incluyen dos relaciones más para describir los datos de los préstamos concedidos en las diferentes sucursales del banco:

$$\begin{aligned} \text{Esquema_préstamo} &= (\text{número_préstamo}, \text{nombre_sucursal}, \text{importe}) \\ \text{Esquema_prestatario} &= (\text{nombre_cliente}, \text{número_préstamo}) \end{aligned}$$

Las Figuras 2.6 y 2.7, respectivamente, muestran las relaciones de ejemplo *préstamo* (*Esquema_préstamo*) y *prestatario* (*Esquema_prestatario*).

Los esquemas de relación se corresponden con el conjunto de tablas que podrían generarse con el método descrito en el Apartado 1.6. Obsérvese que la relación *cliente* puede contener información sobre clientes que no tengan ni cuenta ni préstamo en el banco. La entidad bancaria aquí descrita servirá como ejemplo principal en este capítulo. Cuando sea necesario, habrá que introducir más esquemas de relaciones para ilustrar casos concretos.

2.1.3 Claves

Es necesario disponer de un modo de especificar la manera en que las tuplas de una relación dada se distingan entre sí. Esto se expresa en términos de sus atributos. Es decir, los valores de los valores de los atributos de una tupla deben ser tales que puedan *identificarla únicamente*. En otras palabras, no

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>
P-11	Collado Mediano	900
P-14	Centro	1.500
P-15	Navacerrada	1.500
P-16	Navacerrada	1.300
P-17	Centro	1.000
P-23	Moralzarzal	2.000
P-93	Becerril	500

Figura 2.6 La relación *préstamo*.

se permite que dos tuplas de una misma relación tengan exactamente los mismos valores en todos sus atributos.

Una **superclave** es un conjunto de uno o varios atributos que, considerados conjuntamente, permiten identificar de manera única una tupla de la relación. Por ejemplo, el atributo *id_cliente* de la relación *cliente* es suficiente para distinguir una tupla *cliente* de otra. Por tanto, *id_cliente* es una superclave. De manera parecida, la combinación de *nombre_cliente* e *id_cliente* constituye una superclave para la relación *cliente*. El atributo *nombre_cliente* de *cliente* no es una superclave, ya que es posible que varias personas se llamen igual.

El concepto de superclave no es suficiente para nuestros propósitos, ya que, como se ha podido ver, las superclaves pueden contener atributos innecesarios. Si C es una superclave, entonces también lo es cualquier superconjunto de C. A menudo resultan interesantes superclaves para las que ninguno de sus subconjuntos constituya una superclave. Esas superclaves mínimas se denominan **claves candidatas**.

Es posible que varios conjuntos diferentes de atributos puedan ejercer como claves candidatas. Supóngase que una combinación de *nombre_cliente* y de *calle_cliente* sea suficiente para distinguir entre los miembros de la relación *cliente*. Entonces, tanto {*id_cliente*} como {*nombre_cliente*, *calle_cliente*} son claves candidatas. Aunque los atributos *id_cliente* y *nombre_cliente* en conjunto pueden diferenciar las tuplas *cliente*, su combinación no forma una clave candidata, ya que el atributo *id_cliente* por sí solo ya lo es.

Se usará el término **clave primaria** para denotar una clave candidata que ha elegido el diseñador de la base de datos como medio principal para la identificación de las tuplas de una relación. Las claves (sean primarias, candidatas o superclaves) son propiedades de toda la relación, no de cada una de las tuplas. Ninguna pareja de tuplas de la relación puede tener simultáneamente el mismo valor de los atributos de la clave. La selección de una clave representa una restricción de la empresa del mundo real que se está modelando.

Las claves candidatas deben escogerse con cuidado. Como se ha indicado, el nombre de una persona evidentemente no es suficiente, ya que puede haber mucha gente con el mismo nombre. En Estados Unidos el atributo número de la seguridad social de cada persona sería clave candidata. Dado que los residentes extranjeros no suelen tener número de la seguridad social, las empresas internacionales deben generar sus propios identificadores únicos. Una alternativa es usar como clave alguna combinación exclusiva de otros atributos.

<i>nombre_cliente</i>	<i>número_préstamo</i>
Fernández	P-16
Gómez	P-11
Gómez	P-23
López	P-15
Pérez	P-93
Santos	P-17
Sotoca	P-14
Valdivieso	P-17

Figura 2.7 La relación *prestatario*.

La clave primaria debe escogerse de manera que los valores de sus atributos no se modifiquen nunca, o muy rara vez. Por ejemplo, el campo domicilio de una persona no debe formar parte de la clave primaria, ya que es probable que se modifique. Por otra parte, está garantizado que los números de la seguridad social no cambian nunca. Los identificadores exclusivos generados por las empresas no suelen cambiar, salvo si se produce una fusión entre dos de ellas; en ese caso, puede que el mismo identificador haya sido emitido por ambas empresas, y puede ser necesaria una reasignación de identificadores para garantizar que sean únicos.

Formalmente, sea R el esquema de una relación. Si se dice que un subconjunto C de R es una *superclave* de R , se restringe la consideración a las relaciones $r(R)$ en las que no hay dos tuplas diferentes que tengan los mismos valores en todos los atributos de C . Es decir, si t_1 y t_2 están en r y $t_1 \neq t_2$, entonces $t_1[C] \neq t_2[C]$.

El esquema de una relación, por ejemplo r_1 , puede incluir entre sus atributos la clave primaria de otro esquema de relación, por ejemplo r_2 . Este atributo se denomina **clave externa** de r_1 , que hace referencia a r_2 . La relación r_1 también se denomina **relación referenciante** de la dependencia de clave externa, y r_2 se denomina **relación referenciada** de la clave externa. Por ejemplo, el atributo *nombre_sucursal* de *Esquema_cuenta* es una clave externa de *Esquema_cuenta* que hace referencia a *Esquema_sucursal*, ya que *nombre_sucursal* es la clave primaria de *Esquema_sucursal*. En cualquier ejemplar de la base de datos, dada cualquier tupla, por ejemplo t_a , de la relación *cuenta*, debe haber alguna tupla, por ejemplo t_b , en la relación *sucursal* tal que el valor del atributo *nombre_sucursal* de t_a sea el mismo que el valor de la clave primaria de t_b *nombre_sucursal*.

Es costumbre relacionar los atributos de la clave primaria de un esquema de relación antes que el resto de los atributos; por ejemplo, el atributo *nombre_sucursal* de *Esquema_sucursal* se relaciona en primer lugar, ya que es la clave primaria.

El esquema de la base de datos, junto con las dependencias de clave primaria y externa, se puede mostrar gráficamente mediante **diagramas de esquema**. La Figura 2.8 muestra el diagrama de esquema del ejemplo bancario. Cada relación aparece como un cuadro con los atributos relacionados en su interior y el nombre de la relación sobre él. Si hay atributos de clave primaria, una línea horizontal cruza el cuadro con los atributos de clave primaria por encima de ella y sobre fondo gris. Las dependencias de clave externa aparecen como flechas desde los atributos de clave externa de la relación referenciante a la clave primaria de la relación referenciada.

Muchos sistemas de bases de datos proporcionan herramientas de diseño con una interfaz gráfica de usuario para la creación de los diagramas de esquema.

2.1.4 Lenguajes de consultas

Un **lenguaje de consultas** es un lenguaje en el que los usuarios solicitan información de la base de datos. Estos lenguajes suelen ser de un nivel superior que el de los lenguajes de programación habituales. Los lenguajes de consultas pueden clasificarse como procedimentales o no procedimentales. En los **lenguajes procedimentales** el usuario indica al sistema que lleve a cabo una serie de operaciones en la base

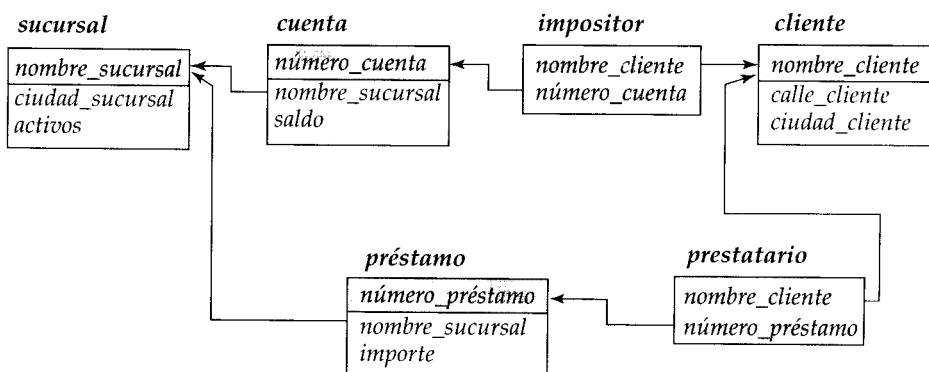


Figura 2.8 Diagrama del esquema de la entidad bancaria.

de datos para calcular el resultado deseado. En los **lenguajes no procedimentales** el usuario describe la información deseada sin dar un procedimiento concreto para obtener esa información.

La mayor parte de los sistemas comerciales de bases de datos relacionales ofrecen un lenguaje de consultas que incluye elementos de los enfoques procedural y no procedural. Se estudiará el muy usado lenguaje de consultas SQL en los Capítulos 3 y 4. El Capítulo 5 trata los lenguajes de consultas QBE y Datalog; este último es un lenguaje de consultas parecido al lenguaje de programación Prolog.

Existen varios lenguajes de consultas “puros”: el álgebra relacional es procedural, mientras que el cálculo relacional de tuplas y el cálculo relacional de dominios no lo son. Estos lenguajes de consultas son rígidos y formales, y carecen del “azúcar sintáctico” de los lenguajes comerciales, pero ilustran las técnicas fundamentales para la extracción de datos de las bases de datos.

En este capítulo se examina con gran detalle el lenguaje del álgebra relacional (en el Capítulo 5 se tratan los lenguajes del cálculo relacional de tuplas y del cálculo relacional de dominios). El álgebra relacional consiste en un conjunto de operaciones que toman una o dos relaciones como entrada y generan otra relación nueva como resultado.

Las operaciones fundamentales del álgebra relacional son *selección*, *proyección*, *unión*, *diferencia de conjuntos*, *producto cartesiano* y *renombramiento*. Además de las operaciones fundamentales hay otras operaciones—por ejemplo, intersección de conjuntos, reunión natural, división y asignación. Estas operaciones se definirán en términos de las operaciones fundamentales.

Inicialmente sólo se estudiarán las consultas. Sin embargo, un lenguaje de manipulación de datos completo no sólo incluye un lenguaje de consultas, sino también un lenguaje para la modificación de las bases de datos. Este tipo de lenguajes incluye comandos para insertar y borrar tuplas, así como para modificar partes de las tuplas existentes. Las modificaciones de las bases de datos se examinarán después de completar la discusión sobre las consultas.

2.2 Operaciones fundamentales del álgebra relacional

Las operaciones selección, proyección y renombramiento se denominan operaciones *unarias* porque operan sobre una sola relación. Las otras tres operaciones operan sobre pares de relaciones y se denominan, por tanto, operaciones *binarias*.

2.2.1 Operación selección

La operación **selección** selecciona tuplas que satisfacen un predicado dado. Se usa la letra griega sigma minúscula (σ) para denotar la selección. El predicado aparece como subíndice de σ . La relación de argumentos se da entre paréntesis a continuación de σ . Por tanto, para seleccionar las tuplas de la relación *préstamo* en la que la sucursal es “Navacerrada” se escribe

$$\sigma_{\text{nombre_sucursal} = \text{"Navacerrada"} } (\text{préstamo})$$

Si la relación *préstamo* es como se muestra en la Figura 2.6, la relación que resulta de la consulta anterior es como aparece en la Figura 2.9.

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>
P-15	Navacerrada	1.500
P-16	Navacerrada	1.300

Figura 2.9 Resultado de $\sigma_{\text{nombre_sucursal} = \text{"Navacerrada"} } (\text{préstamo})$.

Se pueden buscar todas las tuplas en las que el importe prestado sea mayor que 1.200 € escribiendo la siguiente consulta:

$$\sigma_{importe > 1200} (\text{préstamo})$$

En general, se permiten las comparaciones que usan $=, \neq, <, \leq, >$ o \geq en el predicado de selección. Además, se pueden combinar varios predicados en uno mayor usando las conectivas *y* (\wedge), *o* (\vee) y *no* (\neg). Por tanto, para encontrar las tuplas correspondientes a préstamos de más de 1.200 € concedidos por la sucursal de Navacerrada, se escribe

$$\sigma_{nombre_sucursal = "Navacerrada"} \wedge importe > 1200 (\text{préstamo})$$

El predicado de selección puede incluir comparaciones entre dos atributos. Para ilustrarlo, considérese la relación *responsable_préstamo*, que consta de tres atributos: *nombre_cliente*, *nombre_responsable* y *número_préstamo*, que especifica que un empleado concreto es el responsable del préstamo concedido a un cliente. Para hallar todos los clientes que se llaman igual que su responsable de préstamos se puede escribir

$$\sigma_{nombre_cliente = nombre_responsable} (\text{responsable_préstamo})$$

2.2.2 Operación proyección

Supóngase que se desea obtener una relación de todos los números e importes de los préstamos, pero sin los nombres de las sucursales. La operación **proyección** permite obtener esa relación. La operación proyección es una operación unaria que devuelve su relación de argumentos, excluyendo algunos argumentos. Dado que las relaciones son conjuntos, se eliminan todas las filas duplicadas. La proyección se denota por la letra griega mayúscula pi (Π). Se crea una lista de los atributos que se desea que aparezcan en el resultado como subíndice de Π . Su único argumento, una relación, se escribe a continuación entre paréntesis. La consulta para crear una lista de todos los números e importes de los préstamos puede escribirse como

$$\Pi_{número_préstamo, importe} (\text{préstamo})$$

La Figura 2.10 muestra la relación que resulta de esta consulta.

2.2.3 Composición de operaciones relacionales

Es importante el hecho de que el resultado de una operación relacional sea también una relación. Considerese la consulta más compleja “Buscar los clientes que viven en Peguerinos”. Hay que escribir:

$$\Pi_{nombre_cliente} (\sigma_{ciudad_cliente = "Peguerinos"} (\text{cliente}))$$

Téngase en cuenta que, en vez de dar el nombre de una relación como argumento de la operación proyección, se da una expresión cuya evaluación es una relación.

En general, dado que el resultado de las operaciones del álgebra relacional es del mismo tipo (relación) que los datos de entrada, las operaciones del álgebra relacional pueden componerse para formar

<i>número_préstamo</i>	<i>importe</i>
P-11	900
P-14	1.500
P-15	1.500
P-16	1.300
P-17	1.000
P-23	2.000
P-93	500

Figura 2.10 Números e importes de los préstamos.

una **expresión del álgebra relacional**. Componer operaciones del álgebra relacional para formar expresiones del álgebra relacional es igual que componer operaciones aritméticas (como $+$, $-$, $*$ y \div) para formar expresiones aritméticas. La definición formal de las expresiones de álgebra relacional se estudia en el Apartado 2.2.8.

2.2.4 Operación unión

Considérese una consulta para determinar el nombre de todos los clientes del banco que tienen una cuenta, un préstamo o ambas cosas. Obsérvese que la relación *cliente* no contiene esa información, dado que los clientes no necesitan tener ni cuenta ni préstamo en el banco. Para contestar a esta consulta hace falta la información de las relaciones *impositor* (Figura 2.5) y *prestatario* (Figura 2.7). Para determinar los nombres de todos los clientes con préstamos en el banco con las operaciones estudiadas se escribe:

$$\Pi_{\text{nombre_cliente}} (\text{prestatario})$$

Igualmente, para determinar el nombre de todos los clientes con cuenta en el banco se escribe:

$$\Pi_{\text{nombre_cliente}} (\text{impositor})$$

Para contestar a la consulta es necesaria la **unión** de estos dos conjuntos; es decir, hacen falta todos los nombres de clientes que aparecen en alguna de las dos relaciones o en ambas. Estos datos se pueden averiguar mediante la operación binaria unión, denotada, como en la teoría de conjuntos, por \cup . Por tanto, la expresión buscada es:

$$\Pi_{\text{nombre_cliente}} (\text{prestatario}) \cup \Pi_{\text{nombre_cliente}} (\text{impositor})$$

La relación resultante de esta consulta aparece en la Figura 2.11. Téngase en cuenta que en el resultado hay diez tuplas, aunque haya siete prestatarios y seis impositores distintos. Esta discrepancia aparente se debe a que Gómez, López y Santos son a la vez prestatarios e impositores. Dado que las relaciones son conjuntos, se eliminan los valores duplicados.

Obsérvese que en este ejemplo se toma la unión de dos conjuntos, ambos consistentes en valores de *nombre_cliente*. En general, se debe asegurar que las uniones se realicen entre relaciones *compatibles*. Por ejemplo, no tendría sentido realizar la unión de las relaciones *préstamo* y *prestatario*. La primera es una relación con tres atributos, la segunda sólo tiene dos. Más aún, considérese la unión de un conjunto de nombres de clientes y de un conjunto de ciudades. Una unión así no tendría sentido en la mayor parte de los casos. Por tanto, para que la operación unión $r \cup s$ sea válida hay que exigir que se cumplan dos condiciones:

1. Las relaciones r y s deben ser de la misma aridad. Es decir, deben tener el mismo número de atributos.
2. Los dominios de los atributos i -ésimos de r y de s deben ser iguales para todo i .

<i>nombre_cliente</i>
Abril
Fernández
Gómez
González
López
Pérez
Rupérez
Santos
Sotoca
Valdivieso

Figura 2.11 Nombre de todos los clientes que tienen un préstamo o una cuenta.

nombre_cliente
Abril
González
Rupérez

Figura 2.12 Clientes con cuenta abierta pero sin préstamo concedido.

Téngase en cuenta que r y s pueden ser, en general, relaciones de la base de datos o relaciones temporales resultado de expresiones del álgebra relacional.

2.2.5 Operación diferencia de conjuntos

La operación **diferencia de conjuntos**, denotada por $-$, permite hallar las tuplas que están en una relación pero no en la otra. La expresión $r - s$ da como resultado una relación que contiene las tuplas que están en r pero no en s .

Se pueden buscar todos los clientes del banco que tengan abierta una cuenta pero no tengan concedido ningún préstamo escribiendo

$$\Pi_{\text{nombre_cliente}}(\text{impositor}) - \Pi_{\text{nombre_cliente}}(\text{prestatario})$$

La relación resultante de esta consulta aparece en la Figura 2.12.

Como en el caso de la operación unión, hay que asegurarse de que las diferencias de conjuntos se realicen entre relaciones *compatibles*. Por tanto, para que una operación diferencia de conjuntos $r - s$ sea válida se exige que las relaciones r y s sean de la misma aridad y que los dominios de los atributos i-ésimos de r y de s sean iguales.

2.2.6 Operación producto cartesiano

La operación **producto cartesiano**, denotada por un aspa (\times), permite combinar información de cualesquiera dos relaciones. El producto cartesiano de las relaciones r_1 y r_2 se escribe $r_1 \times r_2$.

Recuérdese que las relaciones se definen como subconjuntos del producto cartesiano de un conjunto de dominios. A partir de esa definición ya se debe tener una intuición sobre la definición de la operación producto cartesiano. Sin embargo, dado que el mismo nombre de atributo puede aparecer tanto en r_1 como en r_2 , es necesario crear un convenio de denominación para distinguir unos atributos de otros. En este caso se realiza adjuntando al atributo el nombre de la relación de la que proviene originalmente. Por ejemplo, el esquema de relación de $r = \text{prestatario} \times \text{préstamo}$ es

$$(\text{prestatario.nombre_cliente}, \text{prestatario.número_préstamo}, \\ \text{préstamo.número_préstamo}, \text{préstamo.nombre_sucursal}, \text{préstamo.importe})$$

Con este esquema se puede distinguir entre $\text{prestatario.número_préstamo}$ y $\text{préstamo.número_préstamo}$. Para los atributos que sólo aparecen en uno de los dos esquemas se suele omitir el prefijo con el nombre de la relación. Esta simplificación no genera ambigüedad alguna. Por tanto, se puede escribir el esquema de la relación r como

$$(\text{nombre_cliente}, \text{prestatario.número_préstamo}, \\ \text{préstamo.número_préstamo}, \text{nombre_sucursal}, \text{importe})$$

Este convenio de denominaciones *exige* que las relaciones que sean argumentos de la operación producto cartesiano tengan nombres diferentes. Esta exigencia causa problemas en algunos casos, como cuando se desea calcular el producto cartesiano de una relación consigo misma. Se produce un problema parecido si se usa el resultado de una expresión del álgebra relacional en un producto cartesiano, dado que hará falta un nombre de relación para poder hacer referencia a sus atributos. En el Apartado 2.2.7 se verá la manera de evitar estos problemas mediante la operación renombramiento.

Ahora que se conoce el esquema de relación de $r = \text{prestatario} \times \text{préstamo}$ es necesario hallar las tuplas que aparecerán en r . Como es posible imaginar, se crea una tupla de r a partir de cada par de tuplas

<i>nombre_cliente</i>	<i>prestatario.</i> <i>número_préstamo</i>	<i>préstamo.</i> <i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>
Fernández	P-16	P-11	Collado Mediano	900
Fernández	P-16	P-14	Centro	1.500
Fernández	P-16	P-15	Navacerrada	1.500
Fernández	P-16	P-16	Navacerrada	1.300
Fernández	P-16	P-17	Centro	1.000
Fernández	P-16	P-23	Moralzarzal	2.000
Fernández	P-16	P-93	Becerril	500
Gómez	P-11	P-11	Collado Mediano	900
Gómez	P-11	P-14	Centro	1.500
Gómez	P-11	P-15	Navacerrada	1.500
Gómez	P-11	P-16	Navacerrada	1.300
Gómez	P-11	P-17	Centro	1.000
Gómez	P-11	P-23	Moralzarzal	2.000
Gómez	P-11	P-93	Becerril	500
Gómez	P-23	P-11	Collado Mediano	900
Gómez	P-23	P-14	Centro	1.500
Gómez	P-23	P-15	Navacerrada	1.500
Gómez	P-23	P-16	Navacerrada	1.300
Gómez	P-23	P-17	Centro	1.000
Gómez	P-23	P-23	Moralzarzal	2.000
Gómez	P-23	P-93	Becerril	500
...
...
...
Sotoca	P-14	P-11	Collado Mediano	900
Sotoca	P-14	P-14	Centro	1.500
Sotoca	P-14	P-15	Navacerrada	1.500
Sotoca	P-14	P-16	Navacerrada	1.300
Sotoca	P-14	P-17	Centro	1.000
Sotoca	P-14	P-23	Moralzarzal	2.000
Sotoca	P-14	P-93	Becerril	500
Valdivieso	P-17	P-11	Collado Mediano	900
Valdivieso	P-17	P-14	Centro	1.500
Valdivieso	P-17	P-15	Navacerrada	1.500
Valdivieso	P-17	P-16	Navacerrada	1.300
Valdivieso	P-17	P-17	Centro	1.000
Valdivieso	P-17	P-23	Moralzarzal	2.000
Valdivieso	P-17	P-93	Becerril	500

Figura 2.13 Resultado de *prestatario* \times *préstamo*.

possible: una de la relación *prestatario* y otra de *préstamo*. Por tanto, r es una relación de gran tamaño, como se puede ver en la Figura 2.13, donde sólo se ha incluido una parte de las tuplas que constituyen r .

Supóngase que se tienen n_1 tuplas de *prestatario* y n_2 tuplas de *préstamo*. Por tanto, hay $n_1 * n_2$ maneras de escoger un par de tuplas—una tupla de cada relación; por lo que hay $n_1 * n_2$ tuplas en r . En concreto, obsérvese que para algunas tuplas t de r puede ocurrir que $t[\text{prestatario}.\text{número_préstamo}] \neq t[\text{préstamo}.\text{número_préstamo}]$.

En general, si se tienen las relaciones $r_1(R_1)$ y $r_2(R_2)$, $r_1 \times r_2$ es una relación cuyo esquema es la concatenación de R_1 y de R_2 . La relación R contiene todas las tuplas t para las que hay unas tuplas t_1 en r_1 y t_2 en r_2 para las que $t[R_1] = t_1[R_1]$ y $t[R_2] = t_2[R_2]$.

<i>nombre_cliente</i>	<i>prestatario.número_préstamo</i>	<i>préstamo.número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>
Fernández	P-16	P-15	Navacerrada	1.500
Fernández	P-16	P-16	Navacerrada	1.300
Gómez	P-11	P-15	Navacerrada	1.500
Gómez	P-11	P-16	Navacerrada	1.300
Gómez	P-23	P-15	Navacerrada	1.500
Gómez	P-23	P-16	Navacerrada	1.300
López	P-15	P-15	Navacerrada	1.500
López	P-15	P-16	Navacerrada	1.300
Pérez	P-93	P-15	Navacerrada	1.500
Pérez	P-93	P-16	Navacerrada	1.300
Santos	P-17	P-15	Navacerrada	1.500
Santos	P-17	P-16	Navacerrada	1.300
Sotoca	P-14	P-15	Navacerrada	1.500
Sotoca	P-14	P-16	Navacerrada	1.300
Valdivieso	P-17	P-15	Navacerrada	1.500
Valdivieso	P-17	P-16	Navacerrada	1.300

Figura 2.14 Resultado de $\sigma_{\text{nombre_sucursal} = \text{"Navacerrada"}}(\text{prestatario} \times \text{préstamo})$.

Supóngase que se desea determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada. Se necesita para ello información de las relaciones *préstamo* y *prestatario*. Si se escribe

$$\sigma_{\text{nombre_sucursal} = \text{"Navacerrada}}(\text{prestatario} \times \text{préstamo})$$

entonces el resultado es la relación mostrada en la Figura 2.14. Se tiene una relación que sólo ataña a la sucursal de Navacerrada. Sin embargo, la columna *nombre_cliente* puede contener clientes que no tengan concedido ningún préstamo en la sucursal de Navacerrada. (Si no se ve el motivo por el que esto es cierto, recuérdese que el producto cartesiano toma todos los emparejamientos posibles de cada tupla de *prestatario* con cada tupla de *préstamo*).

Dado que la operación producto cartesiano asocia *todas* las tuplas de *préstamo* con todas las tuplas de *prestatario*, se sabe que, si un cliente tiene concedido un préstamo en la sucursal de Navacerrada, hay alguna tupla de *prestatario* \times *préstamo* que contiene su nombre y que *prestatario.número_préstamo* = *préstamo.número_préstamo*. Por tanto, si se escribe

$$\begin{aligned} \sigma_{\text{prestatario.número_préstamo} = \text{préstamo.número_préstamo}} \\ (\sigma_{\text{nombre_sucursal} = \text{"Navacerrada}}(\text{prestatario} \times \text{préstamo})) \end{aligned}$$

sólo se obtienen las tuplas de *prestatario* \times *préstamo* que corresponden a los clientes que tienen concedido un préstamo en la sucursal de Navacerrada.

Finalmente, dado que sólo se desea obtener *nombre_cliente*, se realiza una proyección:

$$\Pi_{\text{nombre_cliente}} (\sigma_{\text{prestatario.número_préstamo} = \text{préstamo.número_préstamo}} \\ (\sigma_{\text{nombre_sucursal} = \text{"Navacerrada}}(\text{prestatario} \times \text{préstamo})))$$

El resultado de esta expresión, mostrada en la Figura 2.15, es la respuesta correcta a la consulta formulada.

2.2.7 Operación renombramiento

A diferencia de las relaciones de la base de datos, los resultados de las expresiones de álgebra relacional no tienen un nombre que se pueda usar para referirse a ellas. Resulta útil poder ponerles nombre; la operación **renombramiento**, denotada por la letra griega ro minúscula (ρ), permite hacerlo. Dada una

nombre_cliente
Fernández
López

Figura 2.15 Resultado de $\Pi_{\text{nombre_cliente}}$

$$(\sigma_{\text{prestatario.número_préstamo} = \text{préstamo.número_préstamo}} \\ (\sigma_{\text{nombre_sucursal} = \text{"Navacerrada"} (\text{prestatario} \times \text{préstamo}))}).$$

expresión E del álgebra relacional, la expresión

$$\rho_x(E)$$

devuelve el resultado de la expresión E con el nombre x .

Las relaciones r por sí mismas se consideran expresiones (triviales) del álgebra relacional. Por tanto, también se puede aplicar la operación renombramiento a una relación r para obtener la misma relación con un nombre nuevo.

Otra forma de la operación renombramiento es la siguiente. Supóngase que una expresión del álgebra relacional E tiene aridad n . Por tanto, la expresión

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

devuelve el resultado de la expresión E con el nombre x y con los atributos con el nombre cambiado a A_1, A_2, \dots, A_n .

Para ilustrar el renombramiento de relaciones, considérese la consulta “Buscar el saldo máximo de las cuentas del banco”. La estrategia empleada para obtener el resultado es (1) calcular en primer lugar una relación temporal consistente en los saldos que no son el máximo y (2) realizar la diferencia entre la relación $\Pi_{\text{saldo}}(\text{cuenta})$ y la relación temporal recién calculada para obtener el resultado.

Paso 1: para calcular la relación intermedia hay que comparar los valores de los saldos de todas las cuentas. Esta comparación se hará calculando el producto cartesiano $\text{cuenta} \times \text{cuenta}$ y formando una selección para comparar el valor de cualesquiera dos saldos que aparezcan en una tupla. En primer lugar hay que crear un mecanismo para distinguir entre los dos atributos saldo . Se usará la operación renombramiento para cambiar el nombre de una referencia a la relación cuenta ; de este modo se puede hacer referencia dos veces a la relación sin ambigüedad alguna.

La relación temporal que se compone de los saldos que no son el máximo puede escribirse ahora como:

$$\Pi_{\text{cuenta.saldo}} (\sigma_{\text{cuenta.saldo} < d.\text{saldo}} (\text{cuenta} \times \rho_d(\text{cuenta})))$$

Esta expresión proporciona los saldos de la relación cuenta para los que aparece un saldo mayor en alguna parte de la relación cuenta (cuyo nombre se ha cambiado a d). El resultado contiene todos los saldos *salvo* el máximo. La Figura 2.16 muestra esta relación.

Paso 2: la consulta para determinar el saldo máximo de las cuentas del banco puede escribirse de la manera siguiente:

$$\Pi_{\text{saldo}}(\text{cuenta}) - \\ \Pi_{\text{cuenta.saldo}} (\sigma_{\text{cuenta.saldo} < d.\text{saldo}} (\text{cuenta} \times \rho_d(\text{cuenta})))$$

saldo
350
400
500
700
750

Figura 2.16 Resultado de la subexpresión

$$\Pi_{\text{cuenta.saldo}} (\sigma_{\text{cuenta.saldo} < d.\text{saldo}} (\text{cuenta} \times \rho_d(\text{cuenta}))).$$

saldo
900

Figura 2.17 Saldo máximo de las cuentas del banco.

La Figura 2.17 muestra el resultado de esta consulta.

Considérese la consulta “Determinar el nombre de todos los clientes que viven en la misma ciudad y en la misma calle que Gómez” como un nuevo ejemplo de la operación renombramiento. Se puede obtener la calle y la ciudad en las que vive Gómez escribiendo

$$\Pi_{\text{calle_cliente}, \text{ciudad_cliente}} (\sigma_{\text{nombre_cliente} = \text{"Gómez"} }(\text{cliente}))$$

Sin embargo, para hallar a otros clientes que vivan en esa ciudad y en esa calle hay que hacer referencia por segunda vez a la relación *cliente*. En la consulta siguiente se usa la operación renombramiento sobre la expresión anterior para darle al resultado el nombre *dirección_gómez* y para cambiar el nombre de los atributos *calle_cliente* y *ciudad_cliente* a *calle* y *ciudad*, respectivamente:

$$\begin{aligned} & \Pi_{\text{cliente.nombre_cliente}} \\ & (\sigma_{\text{cliente.calle_cliente} = \text{dirección_gómez.calle} \wedge \text{cliente.ciudad_cliente} = \text{dirección_gómez.ciudad}} \\ & (\text{cliente} \times \rho_{\text{dirección_gómez}(\text{calle}, \text{ciudad})} \\ & (\Pi_{\text{calle_cliente}, \text{ciudad_cliente}} (\sigma_{\text{nombre_cliente} = \text{"Gómez"} }(\text{cliente})))))) \end{aligned}$$

El resultado de esta consulta, cuando se aplica a la relación *cliente* de la Figura 2.4, se muestra en la Figura 2.18.

La operación renombramiento no es estrictamente necesaria, dado que es posible usar una notación posicional para los atributos. Se pueden nombrar los atributos de una relación de manera implícita donde \$1, \$2, ... hagan referencia respectivamente al primer atributo, al segundo, etc. La notación posicional también se aplica a los resultados de las operaciones del álgebra relacional. La siguiente expresión del álgebra relacional ilustra el empleo de esta notación con el operador unario σ :

$$\sigma_{\$2=\$3}(R \times R)$$

Si una operación binaria necesita distinguir entre las dos relaciones que son sus operandos, se puede usar una notación posicional parecida para los nombres de las relaciones. Por ejemplo, $\$R1$ puede hacer referencia al primer operando y $\$R2$, al segundo. Sin embargo, la notación posicional no resulta conveniente para las personas, dado que la posición del atributo es un número en vez de un nombre de atributo fácil de recordar. Por tanto, en este libro no se usa la notación posicional.

2.2.8 Definición formal del álgebra relacional

Las operaciones del Apartado 2.2 permiten dar una definición completa de las expresiones del álgebra relacional. Las expresiones fundamentales del álgebra relacional se componen de alguno de los siguientes elementos:

- Una relación de la base de datos
- Una relación constante

Las relaciones constantes se escriben poniendo una relación de sus tuplas entre llaves ($\{\}$), por ejemplo $\{(C-101, Centro, 500) (C-215, Becerril, 700)\}$.

nombre_cliente
Gómez
Pérez

Figura 2.18 Clientes que viven en la misma ciudad y en la misma calle que Gómez.

nombre_cliente
Gómez
López
Santos

Figura 2.19 Clientes con una cuenta abierta y un préstamo en el banco.

Las expresiones generales del álgebra relacional se construyen a partir de subexpresiones más pequeñas. Sean E_1 y E_2 expresiones de álgebra relacional. Todas las expresiones siguientes son también expresiones del álgebra relacional:

- $E_1 \cup E_2$
- $E_1 - E_2$
- $E_1 \times E_2$
- $\sigma_P(E_1)$, donde P es un predicado de atributos de E_1
- $\Pi_S(E_1)$, donde S es una lista que se compone de algunos de los atributos de E_1
- $\rho_x(E_1)$, donde x es el nuevo nombre del resultado de E_1

2.3 Otras operaciones del álgebra relacional

Las operaciones fundamentales del álgebra relacional son suficientes para expresar cualquier consulta del álgebra relacional¹. Sin embargo, limitándose exclusivamente a las operaciones fundamentales, algunas consultas habituales resultan complicadas de expresar. Por tanto, se definen otras operaciones que no añaden potencia al álgebra, pero que simplifican las consultas habituales. Para cada operación nueva se facilita una expresión equivalente usando sólo las operaciones fundamentales.

2.3.1 Operación intersección de conjuntos

La primera operación adicional del álgebra relacional que se va a definir es la **intersección de conjuntos** (\cap). Supóngase que se desea conocer todos los clientes con un préstamo concedido y una cuenta abierta. Empleando la intersección de conjuntos se puede escribir

$$\Pi_{\text{nombre_cliente}}(\text{prestatario}) \cap \Pi_{\text{nombre_cliente}}(\text{impositor})$$

La relación resultante de esta consulta aparece en la Figura 2.19.

Obsérvese que se puede volver a escribir cualquier expresión del álgebra relacional que utilice la intersección de conjuntos sustituyendo la operación intersección por un par de operaciones de diferencia de conjuntos, de la manera siguiente:

$$r \cap s = r - (r - s)$$

Por tanto, la intersección de conjuntos no es una operación fundamental y no añade potencia al álgebra relacional. Sencillamente, es más conveniente escribir $r \cap s$ que $r - (r - s)$.

2.3.2 Operación reunión natural

Suele resultar deseable simplificar ciertas consultas que exijan un producto cartesiano. Generalmente, las consultas que implican un producto cartesiano incluyen un operador selección sobre el resultado del producto cartesiano. Considérese la consulta “Hallar los nombres de todos los clientes que tienen concedido un préstamo en el banco y averiguar su número e importe”. En primer lugar se calcula el

1. En el Apartado 2.4 se introducen operaciones que extienden la potencia del álgebra relacional al tratamiento de los valores nulos y de los valores de agregación.

nombre_cliente	número_préstamo	importe
Fernández	P-16	1.300
Gómez	P-23	2.000
Gómez	P-11	900
López	P-15	1.500
Pérez	P-93	500
Santos	P-17	1.000
Sotoca	P-14	1.500
Valdivieso	P-17	1.000

Figura 2.20 Resultado de $\Pi_{\text{nombre_cliente}, \text{número_préstamo}, \text{importe}} (\text{prestatario} \bowtie \text{préstamo})$.

producto cartesiano de las relaciones *prestatario* y *préstamo*. Después se seleccionan las tuplas que sólo atañen al mismo *número_préstamo*, seguidas por la proyección de *nombre_cliente*, *número_préstamo* e *importe* resultantes:

$$\begin{aligned} & \Pi_{\text{nombre_cliente}, \text{préstamo}, \text{número_préstamo}, \text{importe}} \\ & (\sigma_{\text{prestatario}. \text{número_préstamo} = \text{préstamo}. \text{número_préstamo}} (\text{prestatario} \times \text{préstamo})) \end{aligned}$$

La *reunión natural* es una operación binaria que permite combinar ciertas selecciones y un producto cartesiano en una sola operación. Se denota por el símbolo de **reunión** \bowtie . La operación reunión natural forma un producto cartesiano de sus dos argumentos, realiza una selección forzando la igualdad de los atributos que aparecen en ambos esquemas de relación y, finalmente, elimina los atributos duplicados.

Aunque la definición de la reunión natural es compleja, la operación es sencilla de aplicar. A modo de ilustración, considérese nuevamente el ejemplo “Determinar el nombre de todos los clientes que tienen concedido un préstamo en el banco y averiguar su importe”. Esta consulta puede expresarse usando la reunión natural de la manera siguiente:

$$\Pi_{\text{nombre_cliente}, \text{número_préstamo}, \text{importe}} (\text{prestatario} \bowtie \text{préstamo})$$

Dado que los esquemas de *prestatario* y de *préstamo* (es decir, *Esquema_prestatario* y *Esquema_préstamo*) tienen en común el atributo *número_préstamo*, la operación reunión natural sólo considera los pares de tuplas que tienen el mismo valor de *número_préstamo*. Esta operación combina cada uno de estos pares en una sola tupla en la unión de los dos esquemas (es decir, *nombre_cliente*, *nombre_sucursal*, *número_préstamo*, *importe*). Después de realizar la proyección, se obtiene la relación mostrada en la Figura 2.20.

Considérense dos esquemas de relación *R* y *S*—que son, por supuesto, listas de nombres de atributos. Si se consideran los esquemas como *conjuntos*, en vez de como listas, se pueden denotar los nombres de los atributos que aparecen tanto en *R* como en *S* mediante $R \cap S$, y los nombres de los atributos que aparecen en *R*, en *S* o en ambos mediante $R \cup S$. De manera parecida, los nombres de los atributos que aparecen en *R* pero no en *S* se denotan por $R - S$, mientras que $S - R$ denota los nombres de los atributos que aparecen en *S* pero no en *R*. Obsérvese que las operaciones unión, intersección y diferencia aquí operan sobre conjuntos de atributos, y no sobre relaciones.

Ahora es posible dar una definición formal de la reunión natural. Considérense dos relaciones *r(R)* y *s(S)*. La **reunión natural** de *r* y de *s*, denotada mediante $r \bowtie s$ es una relación del esquema $R \cup S$ definida formalmente de la manera siguiente:

$$r \bowtie s = \Pi_{R \cup S} (\sigma_{r.A_1 = s.A_1 \wedge r.A_2 = s.A_2 \wedge \dots \wedge r.A_n = s.A_n} (r \times s))$$

donde $R \cap S = \{A_1, A_2, \dots, A_n\}$.

Como la reunión natural es fundamental para gran parte de la teoría y de la práctica de las bases de datos relacionales, se ofrecen varios ejemplos de su uso.

- Hallar el nombre de todas las sucursales con clientes que tienen una cuenta abierta en el banco y viven en Peguerinos.

nombre_sucursal
Galapagar
Navacerrada

Figura 2.21 Resultado de

$$\Pi_{\text{nombre_sucursal}}(\sigma_{\text{ciudad_cliente} = \text{"Peguerinos"}} (\text{cliente} \bowtie \text{cuenta} \bowtie \text{impositor})).$$

$$\begin{aligned} & \Pi_{\text{nombre_sucursal}} \\ & (\sigma_{\text{ciudad_cliente} = \text{"Peguerinos"}} (\text{cliente} \bowtie \text{cuenta} \bowtie \text{impositor})) \end{aligned}$$

La relación resultante de esta consulta aparece en la Figura 2.21.

Obsérvese que se ha escrito $\text{cliente} \bowtie \text{cuenta} \bowtie \text{impositor}$ sin añadir paréntesis para especificar el orden en que se deben ejecutar las operaciones reunión natural sobre las tres relaciones. En el caso anterior hay dos posibilidades:

$$\begin{aligned} & (\text{cliente} \bowtie \text{cuenta}) \bowtie \text{impositor} \\ & \text{cliente} \bowtie (\text{cuenta} \bowtie \text{impositor}) \end{aligned}$$

No se ha especificado la expresión deseada, ya que las dos son equivalentes. Es decir, la reunión natural es **asociativa**.

- Hallar todos los clientes que tienen una cuenta abierta y un préstamo concedido en el banco.

$$\Pi_{\text{nombre_cliente}} (\text{prestatario} \bowtie \text{impositor})$$

Obsérvese que en el Apartado 2.3.1 se escribió una expresión para esta consulta usando la intersección de conjuntos. A continuación se repite esa expresión.

$$\Pi_{\text{nombre_cliente}} (\text{prestatario}) \cap \Pi_{\text{nombre_cliente}} (\text{impositor})$$

La relación resultante de esta consulta se mostró anteriormente en la Figura 2.19. Este ejemplo ilustra una característica del álgebra relacional: se pueden escribir varias expresiones del álgebra relacional equivalentes que sean bastante diferentes entre sí.

- Sean $r(R)$ y $s(S)$ relaciones sin atributos en común; es decir, $R \cap S = \emptyset$. (\emptyset denota el conjunto vacío). Por tanto, $r \bowtie s = r \times s$.

La operación **reunión zeta** es una extensión de la operación reunión natural que permite combinar una selección y un producto cartesiano en una sola operación. Considérense las relaciones $r(R)$ y $s(S)$, y sea θ un predicado de los atributos del esquema $R \cup S$. La operación **reunión zeta** $r \bowtie_\theta s$ se define de la manera siguiente:

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

2.3.3 Operación división

La operación **división**, denotada por \div , resulta adecuada para las consultas que incluyen la expresión “para todos”. Supóngase que se desea hallar a todos los clientes que tengan abierta una cuenta en *todas* las sucursales ubicadas en Arganzuela. Se pueden obtener todas las sucursales de Arganzuela mediante la expresión

$$r_1 = \Pi_{\text{nombre_sucursal}} (\sigma_{\text{ciudad_sucursal} = \text{"Arganzuela"}} (\text{sucursal}))$$

La relación resultante de esta expresión aparece en la Figura 2.22.

nombre_sucursal
Centro
Galapagar

Figura 2.22 Resultado de $\Pi_{\text{nombre_sucursal}} (\sigma_{\text{ciudad_sucursal} = \text{"Arganzuela"}} (\text{sucursal}))$.

Se pueden encontrar todos los pares (*nombre_cliente*, *nombre_sucursal*) para los que el cliente tiene una cuenta en una sucursal escribiendo

$$r_2 = \Pi_{\text{nombre_cliente}, \text{nombre_sucursal}} (\text{impostor} \bowtie \text{cuenta})$$

La Figura 2.23 muestra la relación resultante de esta expresión.

Ahora hay que hallar los clientes que aparecen en r_2 con los nombres de *todas* las sucursales de r_1 . La operación que proporciona exactamente esos clientes es la operación división. La consulta se formula escribiendo

$$\begin{aligned} & \Pi_{\text{nombre_cliente}, \text{nombre_sucursal}} (\text{impostor} \bowtie \text{cuenta}) \\ & \div \Pi_{\text{nombre_sucursal}} (\sigma_{\text{ciudad_sucursal} = \text{"Arganzuela}} (\text{sucursal})) \end{aligned}$$

El resultado de esta expresión es una relación que tiene el esquema (*nombre_cliente*) y que contiene la tupla (González).

Formalmente, sean $r(R)$ y $s(S)$ relaciones y $S \subseteq R$; es decir, todos los atributos del esquema S están también en el esquema R . La relación $r \div s$ es una relación del esquema $R - S$ (es decir, del esquema que contiene todos los atributos del esquema R que no están en el esquema S). Una tupla t está en $r \div s$ si y sólo si se cumplen estas dos condiciones:

1. t está en $\Pi_{R-S}(r)$
2. Para cada tupla t_s de s hay una tupla t_r de r que cumple las dos condiciones siguientes:
 - a. $t_r[S] = t_s[S]$
 - b. $t_r[R - S] = t$

Puede resultar sorprendente descubrir que, dados una operación división y los esquemas de las relaciones, se pueda definir la operación división en términos de las operaciones fundamentales. Sean $r(R)$ y $s(S)$ dadas, con $S \subseteq R$:

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

Para comprobar que esta expresión es cierta, obsérvese que $\Pi_{R-S}(r)$ proporciona todas las tuplas t que cumplen la primera condición de la definición de la división. La expresión del lado derecho del operador diferencia de conjuntos,

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

elimina las tuplas que no cumplen la segunda condición de la definición de la división. Esto se logra de la manera siguiente. Considérese $\Pi_{R-S}(r) \times s$. Esta relación está en el esquema R y empareja cada tupla de $\Pi_{R-S}(r)$ con cada tupla de s . La expresión $\Pi_{R-S,S}(r)$ sólo reordena los atributos de r .

Por tanto, $(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ genera los pares de tuplas de $\Pi_{R-S}(r)$ y de s que no aparecen en r . Si una tupla t_j está en

$$\Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

<i>nombre_cliente</i>	<i>nombre_sucursal</i>
Abril	Collado Mediano
Gómez	Becerril
González	Centro
González	Galapagar
López	Navacerrada
Rupérez	Moralzarzal
Santos	Galapagar

Figura 2.23 Resultado de $\Pi_{\text{nombre_cliente}, \text{nombre_sucursal}} (\text{impostor} \bowtie \text{cuenta})$.

nombre_cliente	límite	saldo_crédito
Gómez	2.000	400
López	1.500	1.500
Pérez	2.000	1.750
Santos	6.000	700

Figura 2.24 La relación *información_crédito*.

existe alguna tupla t_s de s que no se combina con la tupla t_j para formar una tupla de r . Por tanto, t_j guarda un valor de los atributos $R - S$ que no aparece en $r \div s$. Estos valores son los que se eliminan de $\Pi_{R-S}(r)$.

2.3.4 Operación asignación

En ocasiones resulta conveniente escribir una expresión del álgebra relacional mediante la asignación de partes de esa expresión a variables de relación temporal. La operación **asignación**, denotada por \leftarrow , actúa de manera parecida a la asignación de los lenguajes de programación. Para ilustrar esta operación, considérese la definición de la división dada en el Apartado 2.3.3. Se puede escribir $r \div s$ como

$$\begin{aligned} temp1 &\leftarrow \Pi_{R-S}(r) \\ temp2 &\leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r)) \\ resultado &= temp1 - temp2 \end{aligned}$$

La evaluación de una asignación no hace que se muestre ninguna relación al usuario. Por el contrario, el resultado de la expresión situada a la derecha de \leftarrow se asigna a la variable relación situada a la izquierda de \leftarrow . Esta variable relación puede usarse en expresiones posteriores.

Con la operación asignación se pueden escribir las consultas como programas secuenciales que constan de en una serie de asignaciones seguida de una expresión cuyo valor se muestra como resultado de la consulta. En las consultas del álgebra relacional la asignación siempre debe hacerse a una variable de relación temporal. Las asignaciones a relaciones permanentes constituyen una modificación de la base de datos. Este asunto se estudiará en el Apartado 2.6. Obsérvese que la operación asignación no añade potencia alguna al álgebra. Resulta, sin embargo, una manera conveniente de expresar las consultas complejas.

2.4 Operaciones del álgebra relacional extendida

Las operaciones básicas del álgebra relacional se han extendido de varias formas. Una sencilla es permitir operaciones aritméticas como parte de las proyecciones. Una importante es permitir *operaciones de agregación*, como el cálculo de la suma de los elementos de un conjunto, o su media. Otra extensión importante es la operación *reunión externa*, que permite a las expresiones del álgebra relacional trabajar con los valores nulos, que modelan la información ausente.

2.4.1 Proyección generalizada

La operación **proyección generalizada** extiende la proyección permitiendo que se utilicen funciones aritméticas en la lista de proyección. La operación proyección generalizada es de la forma

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

donde E es cualquier expresión del álgebra relacional y F_1, F_2, \dots, F_n son expresiones aritméticas que incluyen constantes y atributos del esquema de E . Como caso especial, la expresión aritmética puede ser simplemente un atributo o una constante.

Por ejemplo, supóngase que se dispone de una relación *información_crédito*, como se muestra en la Figura 2.24, que proporciona el límite de crédito y el importe consumido actualmente (*saldo_crédito*). Si se desea determinar el importe disponible por cada persona, se puede escribir la expresión siguiente:

<i>nombre_cliente</i>	<i>crédito_disponible</i>
Gómez	1.600
López	0
Pérez	250
Santos	5.300

Figura 2.25 Resultado de $\Pi_{\text{nombre_cliente}, (\text{límite} - \text{saldo_crédito}) \text{ as } \text{crédito_disponible}} (\text{información_crédito})$.

$$\Pi_{\text{nombre_cliente}, \text{límite} - \text{saldo_crédito}} (\text{información_crédito})$$

El atributo resultante de la expresión $\text{límite} - \text{saldo_crédito}$ no tiene nombre asignado. Se puede aplicar la operación renombramiento al resultado de la proyección generalizada para darle nombre. Como conveniencia notacional, el renombramiento de atributos se puede combinar con la proyección generalizada como se ilustra a continuación:

$$\Pi_{\text{nombre_cliente}, (\text{límite} - \text{saldo_crédito}) \text{ as } \text{crédito_disponible}} (\text{información_crédito})$$

Al segundo atributo de esta proyección generalizada se le ha asignado el nombre *crédito_disponible*. La Figura 2.25 muestra el resultado de aplicar esta expresión a la relación de la Figura 2.24.

2.4.2 Funciones de agregación

Las **funciones de agregación** toman un conjunto de valores y devuelven como resultado un único valor. Por ejemplo, la función de agregación **sum** toma un conjunto de valores y devuelve su suma. Por tanto, la función **sum** aplicada al conjunto

$$\{1, 1, 3, 4, 4, 11\}$$

devuelve el valor 24. La función de agregación **avg** devuelve la media de los valores. Cuando se aplica al conjunto anterior devuelve el valor 4. La función de agregación **count** devuelve el número de elementos del conjunto, y devolvería 6 en el caso anterior. Otras funciones de agregación habituales son **min** y **max**, que devuelven los valores mínimo y máximo del conjunto; en el ejemplo anterior devuelven 1 y 11, respectivamente.

Los conjuntos sobre los que operan las funciones de agregación pueden contener valores repetidos; el orden en el que aparezcan los valores no tiene importancia. Estos conjuntos se denominan **multiconjuntos**. Los conjuntos son un caso especial de los multiconjuntos, en los que sólo hay una copia de cada elemento.

Para ilustrar el concepto de agregación se usará la relación *trabajo_por_horas* descrita en la Figura 2.26, que muestra los empleados a tiempo parcial. Supóngase que se desea determinar la suma total de los sueldos de los empleados del banco a tiempo parcial. La expresión del álgebra relacional para esta consulta es la siguiente:

$$\mathcal{G}_{\text{sum}(\text{sueldo})} (\text{trabajo_por_horas})$$

<i>nombre_empleado</i>	<i>nombre_sucursal</i>	<i>sueldo</i>
Cana	Leganés	1.500
Cascollar	Navacerrada	5.300
Catalán	Leganés	1.600
Díaz	Centro	1.300
Fernández	Navacerrada	1.500
González	Centro	1.500
Jiménez	Centro	2.500
Ribera	Navacerrada	1.300

Figura 2.26 La relación *trabajo_por_horas*.

<i>nombre_empleado</i>	<i>nombre_sucursal</i>	<i>sueldo</i>
Cana Catalán	Leganés	1.500
	Leganés	1.600
Díaz González Jiménez	Centro	1.300
	Centro	1.500
	Centro	2.500
Cascallar Fernández Ribera	Navacerrada	5.300
	Navacerrada	1.500
	Navacerrada	1.300

Figura 2.27 La relación *trabajo_por_horas* después de la agrupación.

El símbolo \mathcal{G} es la letra G en el tipo de letra caligráfico; se lee “G caligráfica”. La operación del álgebra relacional \mathcal{G} significa que se debe aplicar la agregación; y su subíndice, la operación de agregación que se aplica. El resultado de la expresión anterior es una relación con un único atributo, que contiene una sola fila con un valor numérico correspondiente a la suma de los sueldos de todos los trabajadores que trabajan en el banco a tiempo parcial.

Supóngase ahora que se desea determinar la suma total de sueldos de los empleados a tiempo parcial de cada sucursal bancaria, en lugar de determinar la suma total. Para ello hay que dividir la relación *trabajo_por_horas* en **grupos** basados en la sucursal y aplicar la función de agregación a cada grupo. La expresión siguiente obtiene el resultado deseado usando el operador de agregación \mathcal{G} :

$$_{\textit{nombre_sucursal}} \mathcal{G}_{\text{sum}(\textit{sueldo})} (\textit{trabajo_por_horas})$$

Existen casos en los que es necesario borrar los valores repetidos antes de calcular una función de agregación. Si se desea eliminar los valores repetidos hay que usar los mismos nombres de funciones que antes, con la cadena de texto “**distinct**” precedida de un guión añadida al final del nombre de la función (por ejemplo, **count-distinct**). Un ejemplo se da en la consulta “Determinar el número de sucursales que aparecen en la relación *trabajo_por_horas*”. En este caso, el nombre de cada sucursal sólo se cuenta una vez, independientemente del número de empleados que trabajen en la misma. Esta consulta se escribe de la manera siguiente:

$$\mathcal{G}_{\text{count}-\text{distinct}(\textit{nombre_sucursal})} (\textit{trabajo_por_horas})$$

Al aplicar esta consulta a la relación de la Figura 2.26 el resultado está compuesto por una única fila con valor 3. En esta consulta el atributo *nombre_sucursal* en el subíndice a la izquierda de \mathcal{G} indica que la relación de entrada *trabajo_por_horas* debe dividirse en grupos de acuerdo con el valor de *nombre_sucursal*. La Figura 2.27 muestra los grupos resultantes. La expresión $\text{sum}(\textit{sueldo})$ en el subíndice a la derecha de \mathcal{G} indica que, para cada grupo de tuplas (es decir, para cada sucursal) hay que aplicar la función de agregación **sum** al conjunto de valores del atributo *sueldo*. La relación resultante consiste en tuplas con el nombre de la sucursal y la suma de los sueldos de la sucursal, como se muestra en la Figura 2.28.

La forma general de la **operación de agregación** \mathcal{G} es la siguiente:

$$_{G_1, G_2, \dots, G_n} \mathcal{G}_{F_1(A_1), F_2(A_2), \dots, F_m(A_m)} (E)$$

<i>nombre_sucursal</i>	<i>sum(sueldo)</i>
Centro	5.300
Leganés	3.100
Navacerrada	8.100

Figura 2.28 Resultado de $_{\textit{nombre_sucursal}} \mathcal{G}_{\text{sum}(\textit{sueldo})} (\textit{trabajo_por_horas})$.

nombre_sucursal	suma_sueldo	suelo_máximo
Centro	5.300	2.500
Leganés	3.100	1.600
Navacerrada	8.100	5.300

Figura 2.29 Resultado de $\text{nombre_sucursal} \mathcal{G}_{\text{sum}(sueldo)} \text{ as } \text{suma_sueldo}, \mathcal{G}_{\text{max}(sueldo)} \text{ as } \text{suelo_máximo} (\text{trabajo_por_horas})$.

donde E es cualquier expresión del álgebra relacional; G_1, G_2, \dots, G_n constituye una lista de atributos sobre los que se realiza la agrupación, cada F_i es una función de agregación y cada A_i es el nombre de un atributo. El significado de la operación es el siguiente: las tuplas del resultado de la expresión E se dividen en grupos de manera que

1. Todas las tuplas de cada grupo tienen los mismos valores para G_1, G_2, \dots, G_n .
2. Las tuplas de grupos diferentes tienen valores diferentes para G_1, G_2, \dots, G_n .

Por tanto, los grupos pueden identificarse por el valor de los atributos G_1, G_2, \dots, G_n . Para cada grupo (g_1, g_2, \dots, g_n) el resultado tiene una tupla $(g_1, g_2, \dots, g_n, a_1, a_2, \dots, a_m)$ donde, para cada i , a_i es el resultado de aplicar la función de agregación F_i al multiconjunto de valores del atributo A_i en el grupo.

Como caso especial de la operación de agrupación, la lista de atributos G_1, G_2, \dots, G_n puede estar vacía, en cuyo caso sólo hay un grupo que contiene todas las tuplas de la relación. Esto se corresponde con la agrupación sin agrupación.

Volviendo al ejemplo anterior, si se desea determinar el sueldo máximo de los empleados a tiempo parcial de cada sucursal, además de la suma de los sueldos, habría que escribir la expresión

$$\text{nombre_sucursal} \mathcal{G}_{\text{sum}(sueldo), \text{max}(sueldo)} (\text{trabajo_por_horas})$$

Como en la proyección generalizada, el resultado de las operaciones de agrupación no tiene nombre. Se puede aplicar la operación renombramiento al resultado para darle un nombre. Como conveniencia notacional, los atributos de las operaciones de agrupación se pueden renombrar como se indica a continuación:

$$\text{nombre_sucursal} \mathcal{G}_{\text{sum}(sueldo) \text{ as } \text{suma_sueldo}, \text{max}(sueldo) \text{ as } \text{suelo_máximo}} (\text{trabajo_por_horas})$$

La Figura 2.29 muestra el resultado de la expresión.

2.4.3 Reunión externa

La operación **reunión externa** es una ampliación de la operación reunión para trabajar con información ausente. Supóngase que se dispone de relaciones con los siguientes esquemas y que contienen datos de empleados a tiempo completo:

$$\begin{aligned} &\text{empleado} (\text{nombre_empleado}, \text{calle}, \text{ciudad}) \\ &\text{trabajo_a_tiempo_completo} (\text{nombre_empleado}, \text{nombre_sucursal}, \text{sueldo}) \end{aligned}$$

Considérense las relaciones *empleado* y *trabajo_a_tiempo_completo* mostradas en la Figura 2.30. Supóngase que se desea generar una única relación con toda la información (calle, ciudad, nombre de la sucursal y sueldo) de los empleados a tiempo completo. Un posible enfoque sería usar la operación reunión natural de la manera siguiente:

$$\text{empleado} \bowtie \text{trabajo_a_tiempo_completo}$$

El resultado de esta expresión se muestra en la Figura 2.31. Obsérvese que se ha perdido la información sobre la calle y la ciudad de residencia de Gómez, dado que la tupla que describe a Gómez no está presente en la relación *trabajo_a_tiempo_completo*; de manera parecida, se ha perdido la información sobre el nombre de la sucursal y sobre el sueldo de Barea, dado que la tupla que describe a Barea no está presente en la relación *empleado*.

nombre_empleado	calle	ciudad
Segura	Tebeo	La Loma
Domínguez	Viaducto	Villaconejos
Gómez	Bailén	Alcorcón
Valdivieso	Fuencarral	Móstoles

nombre_empleado	nombre_sucursal	sueldo
Segura	Majadahonda	1.500
Domínguez	Majadahonda	1.300
Barea	Fuenlabrada	5.300
Valdivieso	Fuenlabrada	1.500

Figura 2.30 Las relaciones *empleado* y *trabajo_a_tiempo_completo*.

nombre_empleado	calle	ciudad	nombre_sucursal	sueldo
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.31 El resultado de *empleado* \bowtie *trabajo_a_tiempo_completo*.

Se puede usar la operación reunión externa para evitar esta pérdida de información. En realidad, esta operación tiene tres formas diferentes: *reunión externa por la izquierda*, denotada por \bowtie ; *reunión externa por la derecha*, denotada por $\bowtie\llcorner$ y *reunión externa completa*, denotada por $\bowtie\bowtie$. Las tres formas de la reunión externa calculan la reunión y añaden tuplas adicionales al resultado de la misma. El resultado de las expresiones *empleado* \bowtie *trabajo_a_tiempo_completo*, *empleado* $\bowtie\llcorner$ *trabajo_a_tiempo_completo* y *empleado* $\bowtie\bowtie$ *trabajo_a_tiempo_completo* se muestra en las Figuras 2.32, 2.33 y 2.34, respectivamente.

La **reunión externa por la izquierda** (\bowtie) toma todas las tuplas de la relación de la izquierda que no coinciden con ninguna tupla de la relación de la derecha, las rellena con valores nulos en todos los demás atributos de la relación de la derecha y las añade al resultado de la reunión natural. En la Figura 2.32 la tupla (Gómez, Bailén, Alcorcón, *nulo*, *nulo*) es una tupla de este tipo. Toda la información de la relación de la izquierda se halla presente en el resultado de la reunión externa por la izquierda.

La **reunión externa por la derecha** ($\bowtie\llcorner$) es simétrica de la reunión externa por la izquierda. Rellena con valores nulos las tuplas de la relación de la derecha que no coinciden con ninguna tupla de la relación de la izquierda y las añade al resultado de la reunión natural. En la Figura 2.33 la tupla (Barea, *nulo*,

nombre_empleado	calle	ciudad	nombre_sucursal	sueldo
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Gómez	Bailén	Alcorcón	<i>nulo</i>	<i>nulo</i>
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.32 Resultado de *empleado* \bowtie *trabajo_a_tiempo_completo*.

nombre_empleado	calle	ciudad	nombre_sucursal	sueldo
Barea	<i>nulo</i>	<i>nulo</i>	Fuenlabrada	5.300
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.33 Resultado de *empleado* $\bowtie\llcorner$ *trabajo_a_tiempo_completo*.

<i>nombre_empleado</i>	<i>calle</i>	<i>ciudad</i>	<i>nombre_sucursal</i>	<i>sueldo</i>
Barea	<i>nulo</i>	<i>nulo</i>	Fuenlabrada	5.300
Domínguez	Viaducto	Villaconejos	Majadahonda	1.300
Gómez	Bailén	Alcorcón	<i>nulo</i>	<i>nulo</i>
Segura	Tebeo	La Loma	Majadahonda	1.500
Valdivieso	Fuencarral	Móstoles	Fuenlabrada	1.500

Figura 2.34 Resultado de $empleado \bowtie_{\text{trabajo_a_tiempo_parcial}}$.

nulo, Fuenlabrada, 5.300) es una tupla de este tipo. Por tanto, toda la información de la relación de la derecha se halla presente en el resultado de la reunión externa por la derecha.

La **reunión externa completa** (\bowtie_C) realiza estas dos operaciones, rellenando las tuplas de la relación de la izquierda que no coinciden con ninguna tupla de la relación de la derecha y las tuplas de la relación de la derecha que no coinciden con ninguna tupla de la relación de la izquierda, y añadiéndolas al resultado de la reunión. La Figura 2.34 muestra el resultado de una reunión externa completa.

Puesto que las operaciones de reunión pueden generar resultados que contengan valores nulos, es necesario especificar la manera en que deben manejar estos valores las diferentes operaciones del álgebra relacional. El Apartado 2.5 aborda este problema.

Es interesante observar que las operaciones de reunión externa pueden expresarse mediante las operaciones básicas del álgebra relacional. Por ejemplo, la operación de reunión externa por la izquierda $r \bowtie s$ se puede expresar como:

$$(r \bowtie s) \cup (r - \Pi_R(r \bowtie s)) \times \{(nulo, \dots, nulo)\}$$

donde la relación constante $\{(nulo, \dots, nulo)\}$ se encuentra en el esquema $S - R$.

2.5 Valores nulos

En este apartado se define la forma en que las diferentes operaciones del álgebra relacional tratan los valores nulos y las complicaciones que surgen cuando los valores nulos participan en las operaciones aritméticas o en las comparaciones. Como se verá, a menudo hay varias formas de tratar los valores nulos y, en consecuencia, las siguientes definiciones pueden ser a veces arbitrarias. Las operaciones y las comparaciones con valores nulos se deben evitar siempre que sea posible.

Dado que el valor especial *nulo* indica “valor desconocido o no existente”, cualquier operación aritmética (como $+$, $-$, $*$ y $/$) que incluya valores nulos debe devolver un valor nulo.

De manera parecida, cualquier comparación (como $<$, \leq , $>$, \geq y \neq) que incluya un valor nulo tiene como resultado el valor especial **desconocido**; no se puede decir si el resultado de la comparación es cierto o falso, así que se dice que el resultado es el nuevo valor lógico *desconocido*.

Las comparaciones que incluyen valores nulos pueden aparecer dentro de expresiones booleanas que incluyan las operaciones y (conjunción), o (disyunción) y no (negación). Por tanto, se debe definir la forma en que estas tres operaciones lógicas tratan el valor lógico *desconocido*.

- **Y:** $(cierto \text{ y } desconocido) = desconocido; (falso \text{ y } desconocido) = falso; (desconocido \text{ y } desconocido) = desconocido.$
- **O:** $(cierto \text{ o } desconocido) = cierto; (falso \text{ o } desconocido) = desconocido; (desconocido \text{ o } desconocido) = desconocido.$
- **No:** $(\text{no } desconocido) = desconocido.$

Ahora es posible describir la forma en que las diferentes operaciones del álgebra relacional tratan los valores nulos.

- **Selección.** La operación selección evalúa el predicado P en $\sigma_P(E)$ sobre cada tupla t de E . Si el predicado devuelve el valor *cierto*, se añade t al resultado. En caso contrario, si el predicado devuelve *desconocido* o *falso*, t no se añade al resultado.

- **Reunión.** Las reuniones se pueden expresar como un producto cartesiano seguido de una selección. Por tanto, la definición de la forma en que la selección trata los nulos también define la forma en que lo hacen las operaciones reunión.

En una reunión natural, por ejemplo, $r \bowtie s$ se puede observar de la definición anterior que si dos tuplas, $t_r \in r$ y $t_s \in s$, tienen un valor nulo en un atributo común, entonces las tuplas no coinciden.

- **Proyección.** La operación proyección trata los valores nulos como cualesquiera otros valores al eliminar valores duplicados. Así, si dos tuplas del resultado de la proyección son exactamente iguales y las dos tienen valores nulos en los mismos campos, se tratan como duplicados.

Esta decisión es un tanto arbitraria ya que, sin conocer el valor real, no se sabe si los dos valores nulos son duplicados o no.

- **Unión, intersección y diferencia.** Estas operaciones tratan los valores nulos igual que la operación proyección; tratan las tuplas que tienen los mismos valores en todos los campos como duplicados, aunque algunos de los campos tengan valores nulos en ambas tuplas.

Este comportamiento es un tanto arbitrario, especialmente en el caso de la intersección y la diferencia, dado que no se sabe si los valores reales (si existen) representados por los nulos son iguales.

- **Proyección generalizada.** Se describió la manera en que se tratan los nulos en las expresiones al comienzo del Apartado 2.5. Las tuplas duplicadas que contienen valores nulos se tratan como en la operación proyección.

- **Funciones de agregación.** Cuando hay valores nulos en los atributos agregados, las operaciones de agregación los tratan igual que en el caso de la proyección: si dos tuplas son iguales en todos los atributos de agregación, la operación las coloca en el mismo grupo, aunque parte de los valores de los atributos sean valores nulos.

Cuando aparecen valores nulos en los atributos agregados, la operación borra los valores nulos del resultado antes de aplicar la agregación. Si el multiconjunto resultante está vacío, el resultado agregado es nulo.

Obsérvese que el tratamiento de los valores nulos en este caso es diferente que en las expresiones aritméticas ordinarias; se podría haber definido el resultado de una operación de agregación como nulo si tan sólo uno de los valores agregados fuera nulo. Sin embargo, esto significaría que un único valor desconocido en un gran grupo podría hacer que el resultado agregado sobre el grupo fuese nulo, y se perdería una gran cantidad de información útil.

- **Reunión externa.** Las operaciones de reunión externa se comportan igual que las operaciones de reunión, excepto sobre las tuplas que no aparecen en el resultado de la reunión. Esas tuplas se pueden añadir al resultado (dependiendo de si la operación es \bowtie , $\bowtie\text{C}$ o $\bowtie\text{L}$) llenando con valores nulos.

2.6 Modificación de la base de datos

Hasta ahora se ha centrado la atención en la extracción de información de la base de datos. En este apartado se abordará la manera de añadir, eliminar y modificar información de la base de datos.

Las modificaciones de la base de datos se expresan mediante la operación asignación. Las asignaciones a las relaciones reales de la base de datos se realizan empleando la misma notación que se describió para la asignación en el Apartado 2.3.

2.6.1 Borrado

Las solicitudes de borrado se expresan básicamente igual que las consultas. Sin embargo, en lugar de mostrar las tuplas al usuario, se eliminan de la base de datos las tuplas seleccionadas. Sólo se pueden borrar tuplas enteras; no se pueden borrar valores de atributos concretos. En el álgebra relacional los borrados se expresan mediante

$$r \leftarrow r - E$$

donde r es una relación y E es una consulta del álgebra relacional.

A continuación se muestran varios ejemplos de borrado:

- Borrar todas las cuentas de Gómez.

$$\text{impositor} \leftarrow \text{impositor} - \sigma_{\text{nombre_cliente} = \text{"Gómez"}}(\text{impositor})$$

- Borrar todos los préstamos con importes entre 0 y 50.

$$\text{préstamo} \leftarrow \text{préstamo} - \sigma_{\text{importe} \geq 0 \wedge \text{importe} \leq 50}(\text{préstamo})$$

- Borrar todas las cuentas de las sucursales de Arganzuela

$$\begin{aligned} r_1 &\leftarrow \sigma_{\text{ciudad_sucursal} = \text{"Arganzuela}}(\text{cuenta} \bowtie \text{sucursal}) \\ r_2 &\leftarrow \Pi_{\text{nombre_sucursal}, \text{número_cuenta}, \text{saldo}}(r_1) \\ \text{cuenta} &\leftarrow \text{cuenta} - r_2 \end{aligned}$$

Obsérvese que en el último ejemplo se ha simplificado la expresión mediante la asignación a relaciones temporales (r_1 y r_2).

2.6.2 Inserción

Para insertar datos en una relación hay que especificar la tupla que se va a insertar o escribir una consulta cuyo resultado sea el conjunto de tuplas que se van a insertar. Evidentemente, el valor de los atributos de las tuplas insertadas debe ser miembro del dominio de cada atributo. De manera parecida, las tuplas insertadas deben tener la aridad correcta. El álgebra relacional expresa las inserciones mediante

$$r \leftarrow r \cup E$$

donde r es una relación y E es una expresión del álgebra relacional. La inserción de una sola tupla se expresa haciendo que E sea una relación constante que contiene una tupla.

Supóngase que se desea insertar el hecho de que Gómez tiene 1.200 € en la cuenta C-973 de la sucursal de Navacerrada. Hay que escribir

$$\begin{aligned} \text{cuenta} &\leftarrow \text{cuenta} \cup \{(\text{C-973}, \text{"Navacerrada"}, 1200)\} \\ \text{impositor} &\leftarrow \text{impositor} \cup \{(\text{"Gómez"}, \text{C-973})\} \end{aligned}$$

De forma más general, puede que se desee insertar tuplas según el resultado de una consulta. Supóngase que se desea ofrecer una nueva cuenta de ahorro con 200 € como regalo a todos los clientes con préstamos concedidos en la sucursal de Navacerrada. Se usará el número de préstamo como número de esta cuenta de ahorro. Se escribe:

$$\begin{aligned} r_1 &\leftarrow (\sigma_{\text{nombre_sucursal} = \text{"Navacerrada}}(\text{prestatario} \bowtie \text{préstamo})) \\ r_2 &\leftarrow \Pi_{\text{número_préstamo}, \text{nombre_sucursal}}(r_1) \\ \text{cuenta} &\leftarrow \text{cuenta} \cup (r_2 \times \{(200)\}) \\ \text{impositor} &\leftarrow \text{impositor} \cup \Pi_{\text{nombre_cliente}, \text{número_préstamo}}(r_1) \end{aligned}$$

En lugar de especificar una tupla como se hizo anteriormente, se especifica un conjunto de tuplas que se inserta en las relaciones $cuenta$ e $impositor$. Cada tupla de la relación $cuenta$ tiene un $número_cuenta$ (que es igual que el número de préstamo), un $nombre_sucursal$ (Navacerrada) y el saldo inicial de la nueva cuenta (200 €). Cada tupla de la relación $impositor$ tiene como $nombre_cliente$ el nombre del prestatario al que se le regala la nueva cuenta y el mismo número de cuenta que la correspondiente tupla de $cuenta$.

2.6.3 Actualización

Puede que, en algunas situaciones, se desee modificar un valor de una tupla sin modificar *todos* los valores de esa tupla. Se puede usar el operador proyección generalizada para llevar a cabo esta tarea:

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n}(r)$$

donde cada F_i es, o bien el i -ésimo atributo de r en el caso de que este atributo no se vaya a actualizar o, en caso contrario, una expresión sólo con constantes y atributos de r que proporciona el nuevo valor del atributo. Téngase en cuenta que el esquema de la expresión resultante de la expresión de proyección generalizada debe coincidir con el esquema original de r .

Si se desea seleccionar varias tuplas de r y actualizar sólo esas tuplas, se puede usar la expresión siguiente, donde P denota la condición de selección que escoge las tuplas que hay que actualizar:

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n} (\sigma_P(r)) \cup (r - \sigma_P(r))$$

Para ilustrar el uso de la operación actualización supóngase que se va a realizar el pago de los intereses y que hay que aumentar todos los saldos en un cinco por ciento:

$$\text{cuenta} \leftarrow \Pi_{\text{número_cuenta}, \text{nombre_sucursal}, \text{saldo}} (\text{saldo} * 1.05) \text{ (cuenta)}$$

Supóngase ahora que las cuentas con saldos superiores a 10.000 € reciben un interés del seis por ciento, mientras que el resto recibe un cinco por ciento:

$$\begin{aligned} \text{cuenta} \leftarrow & \Pi_{\text{número_cuenta}, \text{nombre_sucursal}, \text{saldo}} (\text{saldo} * 1.06) \text{ } (\sigma_{\text{saldo} > 10000} \text{ (cuenta)}) \\ & \cup \Pi_{\text{número_cuenta}, \text{nombre_sucursal}, \text{saldo}} (\text{saldo} * 1.05) \text{ } (\sigma_{\text{saldo} \leq 10000} \text{ (cuenta)}) \end{aligned}$$

2.7 Resumen

- El **modelo de datos relacional** se basa en un conjunto de tablas. El usuario del sistema de bases de datos puede consultar esas tablas, insertar tuplas nuevas, borrar tuplas y actualizar (modificar) las tuplas. Hay varios lenguajes para expresar estas operaciones.
- El **álgebra relacional** define un conjunto de operaciones algebraicas que operan sobre las tablas y devuelven tablas como resultado. Estas operaciones se pueden combinar para obtener expresiones que definen las consultas deseadas. El álgebra define las operaciones básicas usadas en los lenguajes de consultas relacionales.
- Las operaciones del álgebra relacional se pueden dividir en:
 - Operaciones básicas.
 - Operaciones adicionales que se pueden expresar en términos de las operaciones básicas.
 - Operaciones extendidas, algunas de las cuales añaden mayor poder expresivo al álgebra relacional.
- Las bases de datos se pueden modificar con la **inserción**, el **borrado** y la **actualización** de tuplas. Se ha usado el álgebra relacional con el **operador asignación** para expresar estas modificaciones.
- El álgebra relacional es un lenguaje rígido y formal que no resulta adecuado para los usuarios ocasionales de los sistemas de bases de datos. Los sistemas comerciales de bases de datos, por tanto, usan lenguajes con más “azúcar sintáctico”. En los Capítulos 3 y 4 se tomará en consideración el lenguaje más influyente, **SQL**, que está basado en el álgebra relacional.

Términos de repaso

- Tabla.
- Relación.
- Variable tupla.
- Dominio atómico.
- Valor nulo.
- Esquema de la base de datos.
- Ejemplar de la base de datos.
- Esquema de la relación.
- Ejemplar de la relación.
- Claves.
- Clave externa.
 - Relación referenciante.
 - Relación referenciada.
- Diagrama de esquema.
- Lenguaje de consultas.
- Lenguaje procedimental.

- Lenguaje no procedimental.
- Álgebra relacional.
- Operaciones del álgebra relacional:
 - Selección (σ).
 - Proyección (Π).
 - Unión (\cup).
 - Diferencia de conjuntos ($-$).
 - Producto cartesiano (\times).
 - Renombramiento (ρ).
- Operaciones adicionales:
 - Intersección de conjuntos (\cap).
 - Reunión natural (\bowtie).
 - División ($/$).
- Operación asignación.
- Operaciones del álgebra relacional extendida:
 - Proyección generalizada (Π').
- Reunión externa.
 - Reunión externa por la izquierda ($\bowtie L$).
 - Reunión externa por la derecha ($\bowtie R$).
 - Reunión externa completa ($\bowtie C$).
- Agregación (G).
- Multiconjuntos.
- Agrupación.
- Valores nulos.
- Valores lógicos:
 - cierto.
 - falso.
 - desconocido.
- Modificación de la base de datos.
 - Borrado.
 - Inserción.
 - Actualización.

Ejercicios prácticos

- 2.1 Considérese la base de datos relacional de la Figura 2.35, en la que las claves primarias están subrayadas. Obténgase una expresión del álgebra relacional para cada una de las consultas siguientes:
- a. Determinar el nombre de todos los empleados que viven en la misma ciudad y en la misma calle que sus jefes.
 - b. Determinar el nombre de todos los empleados de esta base de datos que no trabajan para el Banco Importante.
 - c. Determinar el nombre de todos los empleados que ganan más que cualquier empleado del Banco Pequeño.
- 2.2 Las operaciones de reunión externa amplían la operación reunión natural de manera que las tuplas de las relaciones participantes no se pierdan en el resultado de la reunión. Describáse la manera en que la operación reunión zeta puede ampliarse para que las tuplas de la relación de la izquierda, las de la relación de la derecha o las de ambas relaciones no se pierdan en el resultado de las reuniones zeta.
- 2.3 Considérese la base de datos relacional de la Figura 2.35. Obténgase una expresión del álgebra relacional para cada una de las peticiones siguientes:
- a. Modificar la base de datos de manera que Santos vive ahora en Tres Cantos.
 - b. Dar a todos los jefes de la base de datos un aumento de sueldo del 10 por ciento.

Ejercicios

- 2.4 Describáse las diferencias de significado entre los términos *relación* y *esquema de la relación*.

empleado (nombre_persona, calle, ciudad)
trabaja (nombre_persona, nombre_empresa, sueldo)
empresa (nombre_empresa, ciudad)
jefe (nombre_persona, nombre_jefe)

Figura 2.35 Base de datos relacional para los ejercicios 2.1, 2.3, 2.5, 2.7 y 2.9.

2.5 Considérese la base de datos relacional de la Figura 2.35, en la que las claves primarias están subrayadas. Obténgase una expresión del álgebra relacional para expresar cada una de las consultas siguientes:

- Determinar los nombres de todos los empleados que trabajan para el Banco Importante.
- Determinar el nombre y la ciudad de residencia de todos los empleados que trabajan para el Banco Importante.
- Determinar el nombre, la calle y la ciudad de residencia de todos los empleados que trabajan para el Banco Importante y ganan más de 10.000 € anuales.
- Determinar el nombre de todos los empleados de esta base de datos que viven en la misma ciudad que la compañía para la que trabajan.
- Supóngase que las compañías pueden estar instaladas en varias ciudades. Hállese todas las compañías instaladas en cada ciudad en la que está instalado el Banco Pequeño.

2.6 Considérese la relación de la Figura 2.20, que muestra el resultado de la consulta “Determinar el nombre de todos los clientes que tienen concedido un préstamo en el banco”. Vuélvase a escribir la consulta para incluir no sólo el nombre, sino también la ciudad de residencia de cada cliente. Obsérvese que ahora el cliente Sotoca ya no aparece en el resultado, aunque en realidad tiene un préstamo concedido por el banco.

- Explíquese el motivo de que Sotoca no aparezca en el resultado.
- Supóngase que se desea que Sotoca aparezca en el resultado. ¿Cómo habría que modificar la base de datos para conseguirlo?
- Una vez más, supóngase que se desea que Sotoca aparezca en el resultado. Escríbese una consulta que utilice una reunión externa que cumpla esta condición sin que haya que modificar la base de datos.

2.7 Considérese la base de datos relacional de la Figura 2.35. Obténgase una expresión del álgebra relacional para cada petición:

- Dar a todos los empleados del Banco Importante un aumento de sueldo del 10 por ciento.
- Dar a todos los jefes de la base de datos un aumento de sueldo del 10 por ciento, a menos que el sueldo resultante sea mayor que 100.000 €. En ese caso, dar sólo un aumento del 3 por ciento.
- Borrar todas las tuplas de la relación *trabajo* de los empleados de Banco Pequeño.

2.8 Usando el ejemplo bancario, escríbanse consultas del álgebra relacional para encontrar las cuentas abiertas por más de dos clientes:

- Usando una función de agregación.
- Sin usar funciones de agregación.

2.9 Considérese la base de datos relacional de la Figura 2.35. Obténgase una expresión del álgebra relacional para cada una de las consultas siguientes:

- Determinar la compañía con mayor número de empleados.
- Determinar la compañía con la nómina más reducida.
- Determinar las compañías cuyos empleados ganen un sueldo más elevado, en media, que el sueldo medio del Banco Importante.

2.10 Determínense dos motivos por los que se puedan introducir valores nulos en la base de datos.

2.11 Considérese el siguiente esquema de relación:

$$\begin{aligned} &\textit{empleado}(\underline{\textit{número_empleado}}, \textit{nombre}, \textit{sucursal}, \textit{edad}) \\ &\textit{libros}(\underline{\textit{isbn}}, \textit{título}, \textit{autores}, \textit{editorial}) \\ &\textit{préstamo}(\underline{\textit{número_empleado}}, \underline{\textit{isbn}}, \textit{fecha}) \end{aligned}$$

Escríbanse las consultas siguientes en el álgebra relacional.

- Determinar el nombre de los empleados que han tomado prestados libros editados por McGraw-Hill.

- b. Determinar el nombre de los empleados que han tomado prestados todos los libros editados por McGraw-Hill.
- c. Determinar el nombre de los empleados que han tomado prestados más de cinco libros diferentes editados por McGraw-Hill.
- d. Para cada editorial, determinar el nombre de los empleados que han tomado prestados más de cinco libros de esa editorial.

Notas bibliográficas

E.F. Codd, del Laboratorio de investigación de San José de IBM propuso el modelo relacional a finales de los años sesenta (Codd [1970]). Este trabajo le supuso la obtención del prestigioso Premio Turing de la ACM en 1981 (Codd [1982]).

Siguiendo el trabajo original de Codd se constituyeron varios proyectos de investigación con el objetivo de crear sistemas de bases de datos relacionales prácticos, como System R en el Laboratorio de investigación de IBM de San José, Ingres en la Universidad de California en Berkeley, Query-by-Example en el Centro de investigación de IBM T.J. Watson.

Atzeni y Antonellis [1993] y Maier [1983] son textos dedicados exclusivamente a la teoría del modelo relacional de datos.

La definición original del álgebra relacional está en Codd [1970]. En Codd [1979] se presentan ampliaciones del modelo relacional y explicaciones sobre la incorporación de los valores nulos al álgebra relacional (el modelo RM/T), así como la de las reuniones externas. Codd [1990] es un compendio de los trabajos de E.F. Codd sobre el modelo relacional. Las reuniones externas también se estudian en Date [1993b].

Actualmente están disponibles comercialmente numerosos productos de bases de datos relacionales. Ejemplos de ellos son DB2 de IBM, Oracle, Sybase, Informix y SQL Server de Microsoft. Entre los sistemas de bases de datos relacionales de código abierto están MySQL y PostgreSQL. Ejemplos de productos de bases de datos para uso personal son Access de Microsoft y FoxPro.

SQL

El álgebra relacional descrita en el Capítulo 2 proporciona una notación concisa y formal para la representación de las consultas. Sin embargo, los sistemas de bases de datos comerciales necesitan un lenguaje de consultas más cómodo para el usuario. En este capítulo y en el Capítulo 4 se estudia SQL, el lenguaje de consultas distribuido comercialmente de más influencia. SQL usa una combinación de constructores del álgebra relacional (Capítulo 2) y del cálculo relacional (Capítulo 5).

Aunque se haga referencia al lenguaje SQL como “lenguaje de consultas”, puede hacer mucho más que consultar las bases de datos. Usando SQL es posible además definir la estructura de los datos, modificar los datos de la base de datos y especificar restricciones de seguridad.

No se pretende proporcionar un manual de usuario completo de SQL. En cambio, se presentan los constructores y conceptos fundamentales de SQL. Las distintas implementaciones de SQL pueden diferenciarse en detalles o admitir sólo un subconjunto del lenguaje completo.

3.1 Introducción

IBM desarrolló la versión original de SQL, originalmente denominado Sequel, como parte del proyecto System R a principios de 1970. El lenguaje Sequel ha evolucionado desde entonces y su nombre ha pasado a ser SQL (Structured Query Language, lenguaje estructurado de consultas). Hoy en día, numerosos productos son compatibles con el lenguaje SQL y se ha establecido como *el lenguaje estándar para las bases de datos relacionales*.

En 1986, ANSI (American National Standards Institute, Instituto nacional americano de normalización) e ISO (International Standards Organization, Organización internacional de normalización) publicaron una norma SQL, denominada SQL-86. En 1989 ANSI publicó una extensión de la norma para SQL denominada SQL-89. La siguiente versión de la norma fue SQL-92 seguida de SQL:1999; la versión más reciente es SQL:2003. Las notas bibliográficas proporcionan referencias a esas normas.

El lenguaje SQL tiene varios componentes:

- **Lenguaje de definición de datos (LDD).** El LDD de SQL proporciona comandos para la definición de esquemas de relación, borrado de relaciones y modificación de los esquemas de relación.
- **Lenguaje interactivo de manipulación de datos (LMD).** El LMD de SQL incluye un lenguaje de consultas basado tanto en el álgebra relacional (Capítulo 2) como en el cálculo relacional de tuplas (Capítulo 5). También contiene comandos para insertar, borrar y modificar tuplas.
- **Integridad.** El LDD de SQL incluye comandos para especificar las restricciones de integridad que deben cumplir los datos almacenados en la base de datos. Las actualizaciones que violan las restricciones de integridad se rechazan.
- **Definición de vistas.** El LDD de SQL incluye comandos para la definición de vistas.

- **Control de transacciones.** SQL incluye comandos para especificar el comienzo y el final de las transacciones.
- **SQL incorporado y SQL dinámico.** SQL incorporado y SQL dinámico definen cómo se pueden incorporar instrucciones de SQL en lenguajes de programación de propósito general como C, C++, Java, PL/I, Cobol, Pascal y Fortran.
- **Autorización.** El LDD de SQL incluye comandos para especificar los derechos de acceso a las relaciones y a las vistas.

En este capítulo se presenta una visión general del LMD básico y de las características básicas del LDD de SQL. Esta descripción se basa principalmente en la muy extendida norma SQL-92, pero también se tratan algunas extensiones de las normas SQL:1999 y SQL:2003.

En el Capítulo 4 se ofrece un tratamiento más detallado del sistema de tipos de SQL, de las restricciones de integridad y de las autorizaciones. En ese capítulo también se describen brevemente SQL incorporado y SQL dinámico, incluyendo las normas ODBC y JDBC para la interacción con las bases de datos desde programas escritos en los lenguajes C y Java. En el Capítulo 9 se esbozan las extensiones orientadas a objetos de SQL que se introdujeron en SQL:1999.

Muchos sistemas de bases de datos soportan la mayor parte de la norma SQL-92 y parte de los nuevos constructores de SQL:1999 y SQL:2003, aunque actualmente ninguno soporta todos los constructores nuevos. También se debe tener en cuenta que muchos sistemas de bases de datos no soportan algunas características de SQL-92, y que muchas bases de datos presentan algunas no estándar que no se estudian en este libro. En caso de que alguna característica del lenguaje aquí descrita no funcione en el sistema de bases de datos que se esté utilizando, se debe consultar el manual de usuario para determinar exactamente lo que soporta.

La empresa que se utiliza para los ejemplos de este capítulo y de los posteriores es una entidad bancaria. La Figura 3.1 muestra el esquema relacional que se utiliza en los ejemplos con los atributos que son claves primarias subrayados. Recuérdese que en el Capítulo 2 ya se definió el esquema de una relación R mediante una relación de sus atributos, y posteriormente se definió una relación r del esquema mediante la notación $r(R)$. La notación de la Figura 3.1 omite el nombre del esquema y define el esquema de la relación directamente mediante una relación de sus atributos.

3.2 Definición de datos

El conjunto de relaciones de cada base de datos debe especificarse en el sistema en términos de un lenguaje de definición de datos (LDD). El LDD de SQL no sólo permite la especificación de un conjunto de relaciones, sino también de la información relativa a esas relaciones, incluyendo:

- El esquema de cada relación.
- El dominio de valores asociado a cada atributo.
- Las restricciones de integridad.
- El conjunto de índices que se deben mantener para cada relación.
- La información de seguridad y de autorización de cada relación.
- La estructura de almacenamiento físico de cada relación en el disco.

```
sucursal(nombre_sucursal, ciudad_sucursal, activos)
cliente (nombre_cliente, calle_cliente, ciudad_cliente)
préstamo (número_préstamo, nombre_sucursal, importe)
prestatario (nombre_cliente, número_préstamo)
cuenta (número_cuenta, nombre_sucursal, saldo)
impositor (nombre_cliente, número_cuenta)
```

Figura 3.1 Esquema de la entidad bancaria.

A continuación se analizará la definición de los esquemas y los valores básicos de los dominios; el análisis de las demás características del LDD de SQL se tratará en el Capítulo 4.

3.2.1 Tipos básicos de dominios

La norma SQL soporta gran variedad de tipos de dominio predefinidos, entre ellos:

- **char(*n*)**. Una cadena de caracteres de longitud fija, con una longitud *n* especificada por el usuario. También se puede utilizar la palabra completa **character**.
- **varchar(*n*)**. Una cadena de caracteres de longitud variable con una longitud máxima *n* especificada por el usuario. La forma completa, **character varying**, es equivalente.
- **int**. Un entero (un subconjunto finito de los enteros dependiente de la máquina). La palabra completa, **integer**, es equivalente.
- **smallint**. Un entero pequeño (un subconjunto dependiente de la máquina del tipo de dominio entero).
- **numeric(*p, d*)**. Un número de coma fija, cuya precisión la especifica el usuario. El número está formado por *p* dígitos (más el signo), y de esos *p* dígitos, *d* pertenecen a la parte decimal. Así, **numeric(3,1)** permite que el número 44.5 se almacene exactamente, pero ni 444.5 ni 0.32 se pueden almacenar exactamente en un campo de este tipo.
- **real, double precision**. Números de coma flotante y números de coma flotante de doble precisión, con precisión dependiente de la máquina.
- **float(*n*)**. Un número de coma flotante cuya precisión es, al menos, de *n* dígitos.

En el Apartado 4.1 se tratan otros tipos de dominio.

3.2.2 Definición básica de esquemas en SQL

Las relaciones se definen mediante el comando **create table**:

```
create table r(A1 D1, A2 D2, ..., An Dn,
    ⟨restricción-integridad1⟩,
    ...,
    ⟨restricción-integridadk⟩))
```

donde *r* es el nombre de la relación, cada *A_i* es el nombre de un atributo del esquema de la relación *r* y *D_i* es el tipo de dominio de los valores del dominio del atributo *A_i*. Hay varias restricciones de integridad válidas. En este apartado sólo se estudiarán las de clave primaria (primary key), que adopta la forma:

- **primary key** (*A_{j₁}, A_{j₂}, ..., A_{j_m}*). La especificación de **clave primaria** determina que los atributos *A_{j₁}, A_{j₂}, ..., A_{j_m}* forman la clave primaria de la relación. Los atributos de la clave primaria tienen que ser *no nulos* y *únicos*; es decir, ninguna tupla puede tener un valor nulo para un atributo de la clave primaria y ningún par de tuplas de la relación puede ser igual en todos los atributos de la clave primaria¹. Aunque la especificación de clave primaria es opcional, suele ser una buena idea especificar una clave primaria para cada relación.

Otras restricciones de integridad que puede incluir el comando **create table** se tratan más adelante, en el Apartado 4.2.

La Figura 3.2 presenta una definición parcial en el LDD de SQL de la base de datos bancaria. Obsérvese que, al igual que en capítulos anteriores, no se intenta modelar con precisión el mundo real en el ejemplo de la base de datos bancaria. En el mundo real varias personas pueden llamarse igual, por lo que *nombre_cliente* no sería clave primaria de *cliente*; probablemente se utilizaría un *id_cliente* como clave primaria.

1. En SQL-89, los atributos que forman la clave primaria no estaban declarados implícitamente como **not null**; era necesaria una declaración explícita.

```

create table cliente
  (nombre_cliente  char(20),
   calle_cliente   char(30),
   ciudad_cliente char(30),
   primary key (nombre_cliente))

create table sucursal
  (nombre_sucursal char(15),
   ciudad_sucursal char(30),
   activos          numeric(16,2),
   primary key (nombre_sucursal))

create table cuenta
  (número_cuenta  char(10),
   nombre_sucursal char(15),
   saldo           numeric(12,2),
   primary key (número_cuenta))

create table impositor
  (nombre_cliente  char(20),
   número_cuenta  char(10),
   primary key (nombre_cliente, número_cuenta))

```

Figura 3.2 Definición de datos en SQL para parte de la base de datos del banco.

Aquí se utiliza *nombre_cliente* como clave primaria para mantener el esquema de la base de datos sencillo y de pequeño tamaño.

Si una tupla recién insertada o recién modificada de una relación contiene valores nulos para cualquiera de los atributos que forman parte de la clave primaria, o si tiene el mismo valor en ellos que otra tupla de la relación, SQL notifica el error e impide la actualización.

Las relaciones recién creadas están inicialmente vacías. Se puede utilizar el comando **insert** para añadir datos a la relación. Por ejemplo, si se desea añadir el hecho de que hay una cuenta C-9732 en la sucursal de Navacerrada con un saldo de 1.200 €, hay que escribir

```

insert into cuenta
  values ('C-9732', 'Navacerrada', 1200)

```

Los valores se especifican en el orden en que se relacionan los atributos correspondientes en el esquema de la relación. El comando **insert** ofrece una serie de características útiles, y se trata con más detalle en el Apartado 3.10.2.

Se puede utilizar el comando **delete** para borrar tuplas de una relación. El comando

```
delete from cuenta
```

borraría todas las tuplas de la relación *cuenta*. Otras formas del comando **delete** permiten borrar sólo tuplas concretas; el comando **delete** se trata con más detalle en el Apartado 3.10.1.

Para eliminar una relación de una base de datos SQL se utiliza el comando **drop table**. Este comando elimina de la base de datos toda la información de la relación. La instrucción

```
drop table r
```

es una acción más drástica que

```
delete from r
```

La última conserva la relación r , pero borra todas sus tuplas. La primera no sólo borra todas las tuplas de la relación r , sino que también elimina su esquema. Una vez eliminada r , no se puede insertar ninguna tupla en dicha relación, a menos que se vuelva a crear con la instrucción **create table**.

El comando **alter table** se utiliza para añadir atributos a una relación existente. Se asigna a todas las tuplas de la relación un valor *nulo* como valor del atributo nuevo. La forma del comando **alter table** es

alter table r add $A D$

donde r es el nombre de una relación ya existente, A es el nombre del atributo que se desea añadir y D es el dominio del atributo añadido. Se pueden eliminar atributos de una relación utilizando el comando

alter table r drop A

donde r es el nombre de una relación ya existente y A es el nombre de un atributo de la relación. Muchos sistemas de bases de datos no permiten el borrado de atributos, aunque sí permiten el borrado de tablas completas.

3.3 Estructura básica de las consultas SQL

Las bases de datos relacionales están formadas por un conjunto de relaciones, a cada una de las cuales se le asigna un nombre único. Cada relación posee una estructura similar a la presentada en el Capítulo 2. SQL permite el uso de valores nulos para indicar que el valor es desconocido o no existe. También permite al usuario especificar los atributos que no pueden contener valores nulos, como se indicó en el Apartado 3.2.

La estructura básica de una expresión SQL consta de tres cláusulas: **select**, **from** y **where**.

- La cláusula **select** se corresponde con la operación proyección del álgebra relacional. Se usa para obtener una relación de los atributos deseados en el resultado de una consulta.
- La cláusula **from** se corresponde con la operación producto cartesiano del álgebra relacional. Genera una lista de las relaciones que deben ser analizadas en la evaluación de la expresión.
- La cláusula **where** se corresponde con el predicado selección del álgebra relacional. Es un predicado que engloba a los atributos de las relaciones que aparecen en la cláusula **from**.

Que el término *select* tenga un significado diferente en SQL que en el álgebra relacional es un hecho histórico desafortunado. En este capítulo se destacan las diferentes interpretaciones para reducir al mínimo las posibles confusiones.

Las consultas habituales de SQL tienen la forma

```

select  $A_1, A_2, \dots, A_n$ 
from  $r_1, r_2, \dots, r_m$ 
where  $P$ 

```

Cada A_i representa un atributo y cada r_i , una relación. P es un predicado. La consulta es equivalente a la expresión del álgebra relacional

$$\Pi_{A_1, A_2, \dots, A_n} (\sigma_P (r_1 \times r_2 \times \dots \times r_m))$$

Si se omite la cláusula **where**, el predicado P es **cierto**. Sin embargo, a diferencia de la expresión del álgebra relacional, el resultado de la consulta SQL puede contener varias copias de algunas tuplas; este aspecto se analizará de nuevo en el Apartado 3.3.8.

SQL forma el producto cartesiano de las relaciones incluidas en la cláusula **from**, lleva a cabo la selección del álgebra relacional utilizando el predicado de la cláusula **where** y después proyecta el resultado sobre los atributos de la cláusula **select**. En la práctica, SQL puede convertir la expresión en una forma equivalente que puede ser procesada de manera más eficiente. No obstante, las cuestiones relativas a la eficiencia se posponen hasta los Capítulos 13 y 14.

3.3.1 La cláusula select

El resultado de las consultas SQL es, por supuesto, una relación. Considérese la consulta simple basada en el ejemplo bancario “Obtener el nombre de todas las sucursales de la relación *préstamo*”:

```
select nombre_sucursal
from préstamo
```

El resultado es una relación consistente en un único atributo con el encabezado *nombre_sucursal*.

Los lenguajes formales de consultas están basados en el concepto matemático de que las relaciones son conjuntos. Así, nunca aparecen tuplas duplicadas en las relaciones. En la práctica, la eliminación de duplicados consume tiempo. Por tanto, SQL (como la mayoría de los lenguajes de consultas comerciales) permite duplicados en las relaciones, así como en el resultado de las expresiones SQL. Así, la consulta anterior mostrará el valor de *nombre_sucursal* una vez por cada tupla de la relación *préstamo* en la que aparezca.

En los casos en los que se desea forzar la eliminación de los valores duplicados, se inserta la palabra clave **distinct** después de **select**. Se puede reescribir la consulta anterior como

```
select distinct nombre_sucursal
from préstamo
```

si se desea eliminar los duplicados.

SQL permite utilizar la palabra clave **all** para especificar de manera explícita que no se eliminen los duplicados:

```
select all nombre_sucursal
from préstamo
```

Dado que la retención de duplicados es la opción predeterminada, no se utilizará **all** en los ejemplos. Para garantizar la eliminación de duplicados en el resultado de los ejemplos de consultas, se usará la cláusula **distinct** siempre que sea necesario. En la mayoría de las consultas en que no se utiliza **distinct**, el número exacto de copias duplicadas de cada tupla presentes en el resultado de la consulta no es importante. Sin embargo, el número sí que es importante en ciertas aplicaciones; esto se volverá a tratar en el Apartado 3.3.8.

El simbolo asterisco “*” se puede usar para denotar “todos los atributos”. Así, el uso de *préstamo.** en la cláusula **select** anterior indicaría que se seleccionarían todos los atributos de *préstamo*. Las cláusulas **select** de la forma **select *** indican que se deben seleccionar todos los atributos de todas las relaciones que aparecen en la cláusula **from**.

La cláusula **select** puede contener también expresiones aritméticas que contengan los operadores +, -, * y / y operen sobre constantes o atributos de la tuplas. Por ejemplo, la consulta

```
select número_préstamo, nombre_sucursal, importe * 100
from préstamo
```

devolverá una relación que es igual que la relación *préstamo*, salvo que el atributo *importe* está multiplicado por 100.

SQL también proporciona tipos de datos especiales, tales como varias formas del tipo *date* (fecha) y permite que varias funciones aritméticas operen sobre esos tipos.

3.3.2 La cláusula where

A continuación se muestra el uso de la cláusula **where** en SQL. Considérese la consulta “Obtener todos los números de préstamo de los préstamos concedidos en la sucursal de Navacerrada e importe superior a 1.200 €”. Esta consulta puede escribirse en SQL como:

```
select número_préstamo
from préstamo
where nombre_sucursal = 'Navacerrada' and importe > 1200
```

SQL usa las conectivas lógicas **and**, **or** y **not** (en lugar de los símbolos matemáticos \wedge , \vee y \neg) en la cláusula **where**. Los operandos de las conectivas lógicas pueden ser expresiones que contengan los operadores de comparación $<$, $<=$, $>$, $>=$, $=$ y $<>$. SQL permite utilizar los operadores de comparación para comparar cadenas y expresiones aritméticas, así como tipos especiales, como los tipos de fecha.

SQL incluye también un operador de comparación **between** para simplificar las cláusulas **where**, el cual especifica que un valor sea menor o igual que un valor y mayor o igual que otro valor. Si se desea obtener el número de préstamo de aquellos préstamos con importe entre 90.000 € y 100.000 €, se puede usar la comparación **between** para escribir

```
select número_préstamo
from préstamo
where importe between 90000 and 100000
```

en lugar de

```
select número_préstamo
from préstamo
where importe  $\geq$  90000 and importe  $\leq$  100000
```

De forma análoga, se puede usar el operador de comparación **not between**.

3.3.3 La cláusula from

Finalmente, se estudiará el uso de la cláusula **from**. La cláusula **from** define por sí misma un producto cartesiano de las relaciones que aparecen en la cláusula. Dado que la reunión natural se define en términos de un producto cartesiano, una selección y una proyección, resulta relativamente sencillo escribir una expresión de SQL para la reunión natural. Se escribe la expresión del álgebra relacional

$$\Pi_{\text{nombre_cliente}, \text{número_préstamo}, \text{importe}} (\text{prestatario} \bowtie \text{préstamo})$$

para la consulta “Para todos los clientes que tienen un préstamo del banco, obtener el nombre, el número de préstamo y su importe”. Esta consulta puede escribirse en SQL como

```
select nombre_cliente, prestatario.número_préstamo, importe
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo
```

Obsérvese que SQL usa la notación *nombre-relación.nombre-atributo*, como lo hace el álgebra relacional, para evitar ambigüedades en los casos en los que un atributo aparece en más de un esquema de relación. También se podría haber usado *prestatario.nombre_cliente* en lugar de *nombre_cliente* en la cláusula **select**. Sin embargo, como el atributo *nombre_cliente* sólo aparece en una de las relaciones de la cláusula **from**, no existe ambigüedad al escribir *nombre_cliente*.

Se puede extender la consulta anterior y considerar un caso más complicado en el que se también se exija que el préstamo se haya concedido en la sucursal de Navacerrada: “Determinar el nombre, el número de préstamo y su importe de todos los préstamos de la sucursal de Navacerrada”. Para escribir esta consulta hay que definir dos restricciones en la cláusula **where**, enlazadas con la conectiva lógica **and**:

```
select nombre_cliente, prestatario.número_préstamo, importe
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo and
    nombre_sucursal = 'Navacerrada'
```

SQL incluye extensiones para llevar a cabo reuniones naturales y reuniones externas en la cláusula **from**. Estas extensiones se estudian en el Apartado 3.11.

3.3.4 La operación renombramiento

SQL proporciona un mecanismo para renombrar tanto relaciones como atributos. Utiliza la cláusula **as** de la forma siguiente:

nombre-antiguo as nombre-nuevo

La cláusula **as** puede aparecer tanto en la cláusula **select** como en la cláusula **from**.

Considérese de nuevo la consulta anterior:

```
select nombre_cliente, prestatario.número_préstamo, importe
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo
```

El resultado de esta consulta es una relación con los atributos siguientes:

nombre_cliente, número_préstamo, importe

Los nombres de los atributos en el resultado proceden de los nombres de los atributos de las relaciones que aparecen en la cláusula **from**.

Sin embargo, no se pueden obtener siempre los nombres de este modo por varios motivos. En primer lugar, dos relaciones de la cláusula **from** pueden tener atributos con el mismo nombre, en cuyo caso, un nombre de atributo aparecerá duplicado en el resultado. En segundo lugar, si se ha utilizado una expresión aritmética en la cláusula **select**, los atributos resultantes no tienen nombre. En tercer lugar, incluso si el nombre de un atributo se puede obtener a partir de las relaciones base, como en el ejemplo anterior, puede que se desee cambiar el nombre del atributo en el resultado. Por ello, SQL proporciona una forma de renombrar los atributos de las relaciones resultantes.

Por ejemplo, si se desea cambiar el nombre del atributo *número_préstamo* por *id_préstamo*, se puede volver a escribir la consulta anterior como

```
select nombre_cliente, prestatario.número_préstamo as id_préstamo, importe
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo
```

3.3.5 Variables tupla

La cláusula **as** resulta especialmente útil en la definición del concepto de variable tupla. Las variables tupla en SQL se deben asociar con una relación concreta. Las variables tupla se definen en la cláusula **from** mediante la cláusula **as**. Para ilustrarlo, se reescribirá la consulta “Para todos los clientes que tienen concedido un préstamo por el banco, obtener el nombre, el número de préstamo y su importe” como

```
select nombre_cliente, T.número_préstamo, S.importe
from prestatario as T, préstamo as S
where T.número_préstamo = S.número_préstamo
```

Obsérvese que la variable tupla se define en la cláusula **from** colocándola después del nombre de la relación a la cual está asociada y detrás de la palabra clave **as** (la palabra clave **as** es opcional). Al escribir expresiones de la forma *nombre_relación.nombre_atributo*, el nombre de la relación es, en efecto, una variable tupla definida implícitamente.

Las variables tupla son de gran utilidad para comparar dos tuplas de la misma relación. Hay que recordar que, en estos casos, se puede usar la operación renombramiento del álgebra relacional. Supóngase que se desea formular la consulta “Determinar el nombre de todas las sucursales que tienen activos mayores que, al menos, una sucursal de Arganzuela”. Se puede escribir la expresión SQL siguiente:

```
select distinct T.nombre_sucursal
from sucursal as T, sucursal as S
where T.activos > S.activos and S.ciudad_sucursal = 'Arganzuela'
```

Obsérvese que no se puede utilizar la notación *sucursal.activos*, puesto que no estaría claro qué referencia a *sucursal* se pretende hacer.

SQL permite usar la notación (v_1, v_2, \dots, v_n) para designar una tupla de aridad n que contiene los valores v_1, v_2, \dots, v_n . Los operadores de comparación se pueden utilizar sobre las tuplas, y el orden se define lexicográficamente. Por ejemplo, $(a_1, a_2) \leq (b_1, b_2)$ es cierto si $a_1 < b_1$, o si se cumple que $(a_1 = b_1) \wedge (a_2 \leq b_2)$; de manera parecida, dos tuplas son iguales si lo son todos sus atributos.

3.3.6 Operaciones con cadenas de caracteres

SQL especifica las cadenas de caracteres encerrándolas entre comillas simples, como en 'Navacerrada', como ya se ha visto anteriormente. Los caracteres de comillas que formen parte de una cadena de caracteres se pueden especificar usando dos caracteres de comillas; por ejemplo, la cadena de caracteres "Peter O'Toole es un gran actor" se puede especificar como "Peter O"Toole es un gran actor".

La operación más utilizada sobre las cadenas de caracteres es la comparación de patrones, para la que se usa el operador **like**. Para la descripción de los patrones se utilizan dos caracteres especiales:

- Tanto por ciento (%). El carácter % coincide con cualquier subcadena de caracteres.
- Subrayado (_). El carácter _ coincide con cualquier carácter.

Los patrones distinguen la caja; es decir, los caracteres en mayúsculas se consideran diferentes de los caracteres en minúscula, y viceversa. Para ilustrar la comparación de patrones, se considerarán los siguientes ejemplos:

- 'Nava %' coincide con cualquier cadena de caracteres que empiece con "Nava".
- '%cer %' coincide con cualquier cadena que contenga "cer" como subcadena, por ejemplo 'Navacerrada', 'Cáceres' y 'Becerril'.
- '_--' coincide con cualquier cadena que tenga exactamente tres caracteres.
- '_--%' encaja con cualquier cadena que tenga, al menos, tres caracteres.

Los patrones se expresan en SQL utilizando el operador de comparación **like**. Considérese la consulta "Determinar el nombre de todos los clientes cuya dirección contenga la subcadena de caracteres 'Mayor'". Esta consulta se puede escribir como

```
select nombre_cliente
from cliente
where calle_cliente like '%Mayor%'
```

Para que los patrones puedan contener los caracteres especiales de patrón (esto es, % y _) SQL permite la especificación de un carácter de escape. El carácter de escape se utiliza inmediatamente antes de los caracteres especiales de patrón para indicar que ese carácter especial se va a tratar como un carácter normal. El carácter de escape para una comparación **like** se define utilizando la palabra clave **escape**. Para ilustrar esto, considérense los siguientes patrones, que utilizan una barra invertida (\) como carácter de escape:

- **like 'ab\ %cd %'** **escape '\'** coincide con todas las cadenas que empiecen por "ab %cd".
- **like 'ab\\cd %'** **escape '\\'** coincide con todas las cadenas que empiecen por "ab\cd".

SQL permite buscar discordancias en lugar de concordancias utilizando el operador de comparación **not like**.

SQL también permite varias funciones que operan sobre cadenas de caracteres, tales como la concatenación (utilizando “||”), la extracción de subcadenas de caracteres, el cálculo de la longitud de las cadenas de caracteres, la conversión a mayúsculas (utilizando **upper()**) y minúsculas (utilizando **lower()**), etc. SQL:1999 también ofrece una operación **similar to**, que proporciona una comparación de patrones más potente que la operación **like**; la sintaxis para especificar patrones es parecida a la usada para las expresiones regulares de Unix.

Existen variaciones en el conjunto concreto de funciones para cadenas de caracteres que soportan los diferentes sistemas de bases de datos. Algunos sistemas de bases de datos no distinguen entre mayúsculas y minúsculas al comparar cadenas de caracteres. Así, 'ABC' **like** 'abc' daría como resultado cierto, igual que 'ABC' = 'abc', en esos sistemas. Otros ofrecen extensiones para especificar que la comparación de cadenas de caracteres debe ignorar si están en mayúsculas o en minúsculas. Conviene leer el manual de cada sistema de bases de datos para conocer más detalles sobre las funciones de cadenas de caracteres que soporta exactamente.

3.3.7 Orden en la presentación de las tuplas

SQL ofrece al usuario cierto control sobre el orden en el cual se presentan las tuplas de una relación. La cláusula **order by** hace que las tuplas del resultado de una consulta se presenten en un cierto orden. Para obtener una relación en orden alfabético de todos los clientes que tienen un préstamo en la sucursal de Navacerrada hay que escribir:

```
select distinct nombre_cliente
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo and
      nombre_sucursal = 'Navacerrada'
order by nombre_cliente
```

De manera predeterminada, la cláusula **order by** coloca los elementos en orden ascendente. Para especificar el tipo de ordenación se puede especificar **desc** para orden descendente o **asc** para orden ascendente. Además, se puede ordenar con respecto a más de un atributo. Supóngase que se desea colocar toda la relación *préstamo* en orden descendente de *importe*. Si varios préstamos tienen el mismo importe, se ordenan de manera ascendente según sus números de préstamo. Esta consulta en SQL se expresa del modo siguiente:

```
select *
from préstamo
order by importe desc, número_préstamo asc
```

SQL debe llevar a cabo una ordenación para evaluar **order by**. Como ordenar un gran número de tuplas puede resultar costoso, sólo debería hacerse cuando sea estrictamente necesario.

3.3.8 Duplicados

El uso de relaciones con duplicados presenta ventajas en diferentes situaciones. En consecuencia, SQL no sólo define las tuplas que están en el resultado de una consulta, sino también el número de copias de cada una de esas tuplas que aparece en el resultado. La semántica de duplicados de una consulta SQL se define utilizando versiones *multiconjunto* de los operadores relacionales. A continuación se definen las versiones multiconjunto de varios de los operadores del álgebra relacional. Dadas las relaciones multiconjunto r_1 y r_2 ,

1. Si existen c_1 copias de la tupla t_1 en r_1 , y t_1 satisface la selección σ_θ , entonces hay c_1 copias de t_1 en $\sigma_\theta(r_1)$.
2. Para cada copia de la tupla t_1 en r_1 , hay una copia de la tupla $\Pi_A(t_1)$ en $\Pi_A(r_1)$, donde $\Pi_A(t_1)$ denota la proyección de la única tupla t_1 .

3. Si existen c_1 copias de la tupla t_1 en r_1 y c_2 copias de la tupla t_2 en r_2 , entonces hay $c_1 * c_2$ copias de la tupla $t_1 \cdot t_2$ en $r_1 \times r_2$.

Por ejemplo, supóngase que las relaciones r_1 con esquema (A, B) y r_2 con esquema (C) son los multiconjuntos siguientes:

$$r_1 = \{(1, a), (2, a)\} \quad r_2 = \{(2), (3), (3)\}$$

Entonces, $\Pi_B(r_1)$ sería $\{(a), (a)\}$, mientras que $\Pi_B(r_1) \times r_2$ sería

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

Ahora se puede definir cuántas copias de cada tupla aparecen en el resultado de una consulta SQL. Una consulta SQL de la forma

```
select  $A_1, A_2, \dots, A_n$ 
from  $r_1, r_2, \dots, r_m$ 
where  $P$ 
```

es equivalente a la expresión del álgebra relacional

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

que usa las versiones multiconjunto de los operadores relacionales σ , Π y \times .

3.4 Operaciones sobre conjuntos

Las operaciones de SQL **union**, **intersect** y **except** operan sobre relaciones y se corresponden con las operaciones del álgebra relacional \cup , \cap y $-$. Al igual que la unión, la intersección y la diferencia de conjuntos del álgebra relacional, las relaciones que participan en las operaciones han de ser *compatibles*; esto es, deben tener el mismo conjunto de atributos.

A continuación se estudia el modo de formular en SQL varias de las consultas de ejemplo consideradas en el Capítulo 2. Ahora se crearán consultas que incluyan las operaciones **union**, **intersect** y **except** de dos conjuntos: el conjunto de todos los clientes que tienen cuenta en el banco, que puede obtenerse con

```
select nombre_cliente
from impositor
```

y el conjunto de los clientes que tienen concedido un préstamo por el banco, que puede obtenerse con

```
select nombre_cliente
from prestatario
```

A partir de ahora, se hará referencia a las relaciones obtenidas como resultado de las consultas anteriores como i y p , respectivamente.

3.4.1 La operación unión

Para determinar todos los clientes del banco que tienen un préstamo, una cuenta o las dos cosas en el banco, hay que escribir:

```
(select nombre_cliente
from impositor)
union
(select nombre_cliente
from prestatario)
```

A diferencia de la cláusula **select**, la operación **union** (unión) elimina los valores duplicados automáticamente. Así, en la consulta anterior, si un cliente—por ejemplo, Santos—tiene varias cuentas o préstamos (o ambas cosas) en el banco, sólo aparecerá una vez en el resultado.

Si se desea conservar todos los duplicados hay que escribir **union all** en lugar de **union**:

```
(select nombre_cliente
  from impositor)
union all
(select nombre_cliente
  from prestatario)
```

El número de tuplas duplicadas en el resultado es igual al número total de valores duplicados que aparecen en *i* y *p*. Así, si Santos tiene tres cuentas y dos préstamos en el banco, en el resultado aparecerán cinco tuplas con el nombre de Santos.

3.4.2 La operación intersección

Para encontrar todos los clientes que tienen tanto un préstamo como una cuenta en el banco, hay que escribir:

```
(select distinct nombre_cliente
  from impositor)
intersect
(select distinct nombre_cliente
  from prestatario)
```

La operación **intersect** (intersección) elimina los valores duplicados automáticamente. Así, en la consulta anterior, si un cliente—por ejemplo, Santos—tiene varias cuentas o préstamos en el banco, sólo aparecerá una vez en el resultado.

Si se desean conservar todos los valores duplicados hay que escribir **intersect all** en lugar de **intersect**:

```
(select nombre_cliente
  from impositor)
intersect all
(select nombre_cliente
  from prestatario)
```

El número de tuplas duplicadas que aparecen en el resultado es igual al número mínimo de valores duplicados que aparecen en *i* y *p*. Por tanto, si Santos tiene tres cuentas y dos préstamos en el banco, en el resultado de la consulta aparecerán dos tuplas con el nombre de Santos.

3.4.3 La operación excepto

Para determinar todos los clientes que tienen cuenta en el banco pero no tienen ningún préstamo concedido hay que escribir

```
(select distinct nombre_cliente
  from impositor)
except
(select nombre_cliente
  from prestatario)
```

La operación **except** (excepto) elimina los valores duplicados automáticamente. Así, en la consulta anterior, sólo aparecerá en el resultado (exactamente una vez) una tupla con el nombre de Santos si Santos tiene una cuenta en el banco pero no tiene ningún préstamo concedido.

Si se desea conservar todos los valores duplicados hay que escribir **except all** en lugar de **except**:

```
(select nombre_cliente
  from impositor)
except all
(select nombre_cliente
  from prestatario)
```

El número de copias duplicadas de una tupla en el resultado es igual al número de copias duplicadas de dicha tupla en *impositor* menos el número de copias duplicadas de la misma tupla en *prestatario*, siempre que la diferencia sea positiva. Así, si Santos tiene tres cuentas y un préstamo en el banco, en el resultado aparecerán dos tuplas con el nombre de Santos. Si, por el contrario, ese cliente tiene dos cuentas y tres préstamos en el banco, en el resultado no habrá ninguna tupla con el nombre de Santos.

3.5 Funciones de agregación

Las *funciones de agregación* son funciones que toman una colección (un conjunto o multiconjunto) de valores como entrada y devuelven un solo valor. SQL ofrece cinco funciones de agregación incorporadas:

- Media: **avg**
- Mínimo: **min**
- Máximo: **max**
- Total: **sum**
- Recuento: **count**

Los datos de entrada para **sum** y **avg** deben ser una colección de números, pero los otros operadores también pueden operar sobre colecciones de datos de tipo no numérico como las cadenas de caracteres.

A modo de ejemplo, considérese la consulta “Determinar el saldo medio de las cuentas de la sucursal de Navacerrada”. Esta consulta se puede formular del modo siguiente:

```
select avg(saldo)
  from cuenta
 where nombre_sucursal = 'Navacerrada'
```

El resultado de esta consulta es una relación con un único atributo, que contiene una sola tupla con un valor numérico correspondiente al saldo medio de la sucursal de Navacerrada. Opcionalmente se puede dar un nombre al atributo de la relación resultante, utilizando la cláusula **as**.

Existen circunstancias en las cuales sería deseable aplicar las funciones de agregación no sólo a un único conjunto de tuplas sino también a un grupo de conjuntos de tuplas; esto se especifica en SQL mediante la cláusula **group by**. El atributo o atributos especificados en la cláusula **group by** se usan para formar grupos. Las tuplas con el mismo valor en todos los atributos de la cláusula **group by** constituyen un mismo grupo.

Como ejemplo, considérese la consulta “Determinar el saldo medio de las cuentas de cada sucursal”. Esta consulta se formula del modo siguiente:

```
select nombre_sucursal, avg(saldo)
  from cuenta
 group by nombre_sucursal
```

La conservación de los valores duplicados es importante para el cálculo de medias. Supóngase que los saldos de las cuentas en la (pequeña) sucursal de Galapagar son 1.000 €, 3.000 €, 2.000 € y 1.000 €. El saldo medio es $7.000/4 = 1.750,00$ €. Si se eliminan los valores duplicados se obtendría un resultado erróneo ($6.000/3 = 2.000$ €).

A veces se desea tratar toda la relación como un solo grupo. En estos casos no se utiliza la cláusula **group by**. Considérese la consulta “Determinar el saldo medio de todas las cuentas”. Esta consulta se formula del modo siguiente:

```
select avg (saldo)
from cuenta
```

Con frecuencia se usa la función de agregación **count** para contar el número de tuplas de una relación. La notación de SQL para esta función es **count (*)**. Por tanto, para determinar el número de tuplas de la relación *cliente* hay que escribir

```
select count (*)
from cliente
```

Se dan casos en los cuales es necesario eliminar los valores duplicados antes de calcular una función de agregación. Si se desea eliminar los valores duplicados hay que utilizar la palabra clave **distinct** en la expresión de agregación. A modo de ejemplo, considérese la consulta “Determinar el número de impositores de cada sucursal”. En este caso cada impositor sólo se debe contar una vez, independientemente del número de cuentas que pueda tener. La consulta se formula del modo siguiente:

```
select nombre_sucursal, count (distinct nombre_cliente)
from impositor, cuenta
where impositor.número_cuenta = cuenta.número_cuenta
group by nombre_sucursal
```

SQL no permite el uso de **distinct** con **count (*)**. Sí es legal el uso de **distinct** con **max** y **min**, aunque el resultado no cambia. Se puede utilizar la palabra clave **all** en lugar de **distinct** para especificar la conservación de los valores duplicados pero, como **all** es la opción predeterminada, no hace falta hacerlo.

A veces resulta útil establecer una condición que se aplique a los grupos en vez de a las tuplas. Por ejemplo, puede que sólo estemos interesados en las sucursales en las que el saldo medio de las cuentas sea superior a 1.200 €. Esta condición no se aplica a una sola tupla, sino a cada grupo creado por la cláusula **group by**. Para expresar este tipo de consultas se utiliza la cláusula **having** de SQL. SQL aplica los predicados de la cláusula **having** una vez formados los grupos, de modo que se puedan usar las funciones de agregación. Esta consulta se expresa en SQL del modo siguiente:

```
select nombre_sucursal, avg (saldo)
from cuenta
group by nombre_sucursal
having avg (saldo) > 1200
```

Si en la misma consulta aparecen una cláusula **where** y una cláusula **having**, SQL aplica primero el predicado de la cláusula **where**. Las tuplas que satisfacen el predicado de la cláusula **where** se dividen luego en grupos según la cláusula **group by**. Después se aplica la cláusula **having** (si existe) a cada grupo; SQL elimina los grupos que no satisfacen el predicado de la cláusula **having**. La cláusula **select** utiliza los grupos restantes para generar las tuplas del resultado de la consulta.

Para ilustrar el uso conjunto de la cláusula **where** y la cláusula **having** en la misma consulta, considérese la consulta “Determinar el saldo medio de cada cliente que vive en Peguerinos y tiene, como mínimo, tres cuentas”.

```
select impositor.nombre_cliente, avg (saldo)
from impositor, cuenta, cliente
where impositor.número_cuenta = cuenta.número_cuenta and
      impositor.nombre_cliente = cliente.nombre_cliente and
      ciudad_cliente = 'Peguerinos'
group by impositor.nombre_cliente
having count (distinct impositor.número_cuenta) >= 3
```

3.6 Valores nulos

SQL permite el uso de valores *nulos* para indicar la ausencia de información sobre el valor de un atributo.

Se puede utilizar la palabra clave especial **null** en un predicado para comprobar si un valor es nulo. Así, para determinar todos los números de préstamo que aparecen en la relación *préstamo* con valores nulos para *importe* hay que escribir

```
select número_préstamo
from préstamo
where importe is null
```

El predicado **is not null** comprueba la ausencia de valores nulos.

El uso de valores *nulos* en las operaciones aritméticas y de comparación causa algunos problemas. En el Apartado 2.5 se vio cómo se manejan los valores nulos en el álgebra relacional. Ahora se va a describir cómo lo hace SQL.

El resultado de las expresiones aritméticas (que incluyan por ejemplo $+$, $-$, $*$ o $/$) es nulo si cualquiera de los valores de entrada es nulo. SQL trata como **unknown** (desconocido) el resultado de cualquier comparación que implique un valor *nulo* (excepto con **is null** y **is not null**).

Dado que el predicado de las cláusulas **where** puede incluir operaciones booleanas como **and**, **or** y **not** sobre los resultados de las comparaciones, se amplían las definiciones de las operaciones booleanas para que manejen el valor **unknown**, como se describe en el Apartado 2.5.

- **and**. El resultado de *cierto and desconocido* es *desconocido*, *falso and desconocido* es *falso*, mientras que *desconocido and desconocido* es *desconocido*.
- **or**. El resultado de *cierto or desconocido* es *cierto*, *falso or desconocido* es *desconocido*, mientras que *desconocido or desconocido* es *desconocido*.
- **not**. El resultado de **not desconocido** es *desconocido*.

El resultado de una instrucción SQL de la forma:

```
select ... from  $R_1, \dots, R_n$  where  $P$ 
```

contiene (proyecciones de) tuplas de $R_1 \times \dots \times R_n$ para las que el predicado P se evalúa como **cierto**. Si el predicado se evalúa como **falso** o **desconocido** para alguna tupla de $R_1 \times \dots \times R_n$, (la proyección de) esa tupla no se añade al resultado.

SQL también permite comprobar si el resultado de una comparación es *desconocido*, en lugar de *cierto* o *falso*, mediante las cláusulas **is unknown** (es *desconocido*) e **is not unknown** (no es *desconocido*).

La existencia de valores nulos también complica el procesamiento de los operadores de agregación. Por ejemplo, supóngase que algunas tuplas en la relación *préstamo* tienen valor nulo para el atributo *importe*. Considérese la siguiente consulta, que calcula el total de los importes de todos los préstamos:

```
select sum (importe)
from préstamo
```

Entre los valores que se van a sumar en la consulta anterior hay valores nulos, puesto que algunas tuplas tienen valor nulo para *importe*. En lugar de decir que la suma total es *nula*, la norma SQL establece que el operador **sum** debe ignorar los valores nulos de los datos de entrada.

En general, las funciones de agregación tratan los valores nulos según la regla siguiente: todas las funciones de agregación excepto **count (*)** ignoran los valores nulos de la colección de datos de entrada. Como consecuencia de ignorar los valores nulos, la colección de valores de entrada puede estar vacía. El cálculo de **count (*)** de una colección vacía se define como 0 y todas las demás operaciones de agregación devuelven un valor nulo cuando se aplican sobre una colección de datos vacía. El efecto de los valores nulos en algunos de los constructores más complicados de SQL puede ser útil.

En SQL:1999 se introdujo un tipo de datos **Boolean**, que puede tomar los valores **true**, **false** y **unknown**. Las funciones de agregación **some** (algún) y **every** (cada), que significan exactamente lo que se espera de ellas, se pueden aplicar a las colecciones de valores booleanos.

3.7 Subconsultas anidadas

SQL proporciona un mecanismo para anidar subconsultas. Las subconsultas son expresiones **select-from-where** que están anidadas dentro de otra consulta. Una finalidad habitual de las subconsultas es llevar a cabo comprobaciones de pertenencia a conjuntos, hacer comparaciones de conjuntos y determinar cardinalidades de conjuntos. Estos usos se estudian en los apartados siguientes.

3.7.1 Pertenencia a conjuntos

SQL permite comprobar la pertenencia de las tuplas a una relación. La conectiva **in** comprueba la pertenencia a un conjunto, donde el conjunto es la colección de valores resultado de una cláusula **select**. La conectiva **not in** comprueba la no pertenencia a un conjunto.

Como ejemplo considérese de nuevo la consulta “Determinar todos los clientes que tienen tanto un préstamo como una cuenta en el banco”. Anteriormente se escribió esta consulta como la intersección de dos conjuntos: el conjunto de los impositores del banco y el conjunto de los prestatarios del banco. Se puede adoptar el enfoque alternativo consistente en determinar todos los titulares de cuentas del banco que son miembros del conjunto de prestatarios. Claramente, esta formulación genera el mismo resultado que la anterior, pero obliga a formular la consulta usando la conectiva de SQL **in**. Se comienza determinando todos los titulares de cuentas con:

```
(select nombre_cliente  
      from impositor)
```

A continuación, se determinan los clientes que son prestatarios del banco y que aparecen en la lista de titulares de cuentas obtenida en la subconsulta anterior. Esto se consigue anidando la subconsulta en un **select** más externo. La consulta resultante es:

```
select distinct nombre_cliente  
      from prestatario  
   where nombre_cliente in (select nombre_cliente  
                             from impositor)
```

Este ejemplo muestra que en SQL es posible escribir la misma consulta de diversas formas. Esta flexibilidad es de gran utilidad, puesto que permite a los usuarios pensar en las consultas del modo que les parezca más natural. Más adelante se verá que en SQL hay una gran cantidad de redundancia.

En el ejemplo anterior se comprobaba la pertenencia a un conjunto en una relación con un solo atributo. En SQL también es posible comprobar la pertenencia a un conjunto en una relación cualquiera. Así, es posible formular la consulta “Determinar todos los clientes que tienen tanto una cuenta como un préstamo en la sucursal de Navacerrada” de otra manera distinta:

```
select distinct nombre_cliente  
      from prestatario, préstamo  
   where prestatario.número_préstamo = préstamo.número_préstamo and  
         nombre_sucursal = 'Navacerrada' and  
         (nombre_sucursal, nombre_cliente) in  
             (select nombre_sucursal, nombre_cliente  
                  from impositor, cuenta  
               where impositor.número_cuenta = cuenta.número_cuenta)
```

El constructor **not in** se utiliza de manera parecida. Por ejemplo, para encontrar todos los clientes que tienen concedido un préstamo en el banco pero no tienen abierta cuenta, se puede escribir:

```
select distinct nombre_cliente
from prestatario
where nombre_cliente not in (select nombre_cliente
                               from impositor)
```

Los operadores **in** y **not in** también se pueden utilizar sobre conjuntos enumerados. La consulta siguiente selecciona los nombres de los clientes que tienen concedido un préstamo en el banco y cuyos nombres no son ni Santos ni Gómez.

```
select distinct nombre_cliente
from prestatario
where nombre_cliente not in ('Santos', 'Gómez')
```

3.7.2 Comparación de conjuntos

Como ejemplo de la capacidad de comparar conjuntos de las consultas anidadas, considérese la consulta “Determinar el nombre de todas las sucursales que poseen activos mayores que, al menos, una sucursal de Arganzuela”. En el Apartado 3.3.5 se formulaba esta consulta del modo siguiente:

```
select distinct T.nombre_sucursal
from sucursal as T, sucursal as S
where T.activos > S.activos and S.ciudad_sucursal = 'Arganzuela'
```

SQL ofrece, sin embargo, un estilo alternativo de formular la consulta anterior. La expresión: “mayor que, al menos, una” se representa en SQL por **> some**. Este constructor permite volver a escribir la consulta de forma más parecida a la formulación de la consulta en lenguaje natural.

```
select nombre_sucursal
from sucursal
where activos > some (select activos
                        from sucursal
                        where ciudad_sucursal = 'Arganzuela')
```

La subconsulta

```
(select activos
from sucursal
where ciudad_sucursal = 'Arganzuela')
```

genera el conjunto de todos los valores de activos para todas las sucursales situadas en Arganzuela. La comparación **> some** de la cláusula **where** de la cláusula **select** más externa es cierta si el valor del atributo **activos** de la tupla es mayor que, al menos, un miembro del conjunto de todos los valores de activos de las sucursales de Arganzuela.

SQL también permite realizar las comparaciones **< some**, **<= some**, **>= some**, **= some** y **<> some**. Como ejercicio, compruébese que **= some** es idéntico a **in**, mientras que **<> some** no es lo mismo que **not in**. En SQL, la palabra clave **any** es sinónimo de **some**. Las primeras versiones de SQL sólo admitían **any**. Versiones posteriores añadieron la alternativa **some** para evitar la ambigüedad lingüística de la palabra inglesa **any** en su idioma original.

Ahora se va a modificar ligeramente la consulta. Se trata de determinar el nombre de todas las sucursales que tienen activos superiores al de todas las sucursales de Arganzuela. El constructor **> all** se corresponde con la expresión “superiores al de todas”. Usando este constructor se puede escribir la consulta del modo siguiente:

```
select nombre_sucursal
from sucursal
where activos > all (select activos
          from sucursal
          where ciudad_sucursal = 'Arganzuela')
```

Al igual que con **some**, SQL también permite las comparaciones **< all**, **<= all**, **>= all**, **= all** y **<> all**. Como ejercicio, compruébese que **<> all** es lo mismo que **not in**.

Como ejemplo adicional de comparación de conjuntos, considérese la consulta “Determinar la sucursal que tiene el saldo medio máximo”. En SQL las funciones de agregación no se pueden componer. Por tanto, no se puede utilizar **max (avg (...))**. En vez de eso, se puede seguir la siguiente estrategia: se comienza por escribir una consulta que determine todos los saldos medios y luego se anida como subconsulta de una consulta mayor que determina las sucursales en las que el saldo medio es mayor o igual que todos los saldos medios:

```
select nombre_sucursal
from cuenta
group by nombre_sucursal
having avg (saldo) >= all (select avg (saldo)
          from cuenta
          group by nombre_sucursal)
```

3.7.3 Comprobación de relaciones vacías

SQL incluye la posibilidad de comprobar si las subconsultas tienen alguna tupla en su resultado. El constructor **exists** devuelve el valor **true** si su argumento subconsulta no resulta vacía. Mediante el constructor **exists** se puede formular la consulta “Determinar todos los clientes que tienen tanto una cuenta abierta como un préstamo concedido en el banco” de otra forma más:

```
select nombre_cliente
from prestatario
where exists (select *
          from impositor
          where impositor.nombre_cliente = prestatario.nombre_cliente)
```

Mediante el constructor **not exists** se puede comprobar la inexistencia de tuplas en el resultado de las subconsultas. Se puede utilizar el constructor **not exists** para simular la operación de continencia de conjuntos (es decir, superconjuntos). Se puede escribir “la relación *A* contiene a la relación *B*” como “**not exists (B except A)**” (aunque no forma parte de las normas SQL-92 ni SQL:1999 el operador **contains** aparecía en algunos de los primeros sistemas relacionales). Para ilustrar el operador **not exists**, considérese otra vez la consulta “Determinar todos los clientes que tienen una cuenta en todas las sucursales de Arganzuela”. Hay que comprobar para cada cliente si el conjunto de todas las sucursales en las que dicho cliente tiene cuenta contiene al conjunto de todas las sucursales de Arganzuela. Mediante el constructor **except** se puede formular la consulta del modo siguiente:

```
select distinct S.nombre_cliente
from impositor as S
where not exists ((select nombre_sucursal
          from sucursal
          where ciudad_sucursal = 'Arganzuela')
          except
          (select R.nombre_sucursal
          from impositor as T, cuenta as R
          where T.numero_cuenta = R.numero_cuenta and
              S.nombre_cliente = T.nombre_cliente)))
```

En este ejemplo, la subconsulta

```
(select nombre_sucursal
  from sucursal
 where ciudad_sucursal = 'Arganzuela')
```

obtiene todas las sucursales de Arganzuela. La subconsulta

```
(select R.nombre_sucursal
  from impositor as T, cuenta as R
 where T.numero_cuenta = R.numero_cuenta and
       S.nombre_cliente = T.nombre_cliente)
```

obtiene todas las sucursales en las cuales el cliente *S.nombre_cliente* tiene cuenta. Por tanto, el **select** más externo toma cada cliente y comprueba si el conjunto de todas las sucursales en las que dicho cliente tiene cuenta contiene al conjunto de todas las sucursales de Arganzuela.

En las consultas que contienen subconsultas se aplica una regla de visibilidad para las variables tupla. En las subconsultas, de acuerdo con esta regla, sólo se pueden usar variables tupla definidas en la propia subconsulta o en cualquier consulta que la contenga. Si una variable tupla está definida tanto localmente en una subconsulta como globalmente en una consulta que contiene a la subconsulta, se aplica la definición local. Esta regla es análoga a las reglas de visibilidad utilizadas habitualmente para las variables en los lenguajes de programación.

3.7.4 Comprobación de la ausencia de tuplas duplicadas

SQL incluye la posibilidad de comprobar si las subconsultas tienen tuplas duplicadas en su resultado. El constructor **unique** devuelve el valor **true** si la subconsulta que se le pasa como argumento no contiene tuplas duplicadas. Mediante el constructor **unique** se puede formular la consulta “Determinar todos los clientes que tienen, a lo sumo, una cuenta en la sucursal de Navacerrada” del siguiente modo:

```
select T.nombre_cliente
  from impositor as T
 where unique (select R.nombre_cliente
                  from cuenta, impositor as R
                 where T.nombre_cliente = R.nombre_cliente and
                       R.numero_cuenta = cuenta.numero_cuenta and
                           cuenta.nombre_sucursal = 'Navacerrada')
```

La existencia de tuplas duplicadas en una subconsulta se puede comprobar mediante el constructor **not unique**. Para ilustrar este constructor, considérese la consulta “Determinar todos los clientes que tienen, al menos, dos cuentas en la sucursal de Navacerrada”, que se puede formular:

```
select distinct T.nombre_cliente
  from impositor T
 where not unique (select R.nombre_cliente
                  from cuenta, impositor as R
                 where T.nombre_cliente = R.nombre_cliente and
                       R.numero_cuenta = cuenta.numero_cuenta and
                           cuenta.nombre_sucursal = 'Navacerrada')
```

Formalmente, la evaluación de **unique** sobre una relación se define como falsa si y sólo si la relación contiene dos tuplas t_1 y t_2 tales que $t_1 = t_2$. Como la comprobación $t_1 = t_2$ es falsa si algún campo de t_1 o de t_2 es nulo, es posible que el resultado de **unique** sea cierto aunque haya varias copias de una misma tupla, siempre que, al menos, uno de los atributos de la tupla sea nulo.

3.8 Consultas complejas

Las consultas complejas a menudo resultan difíciles o imposibles de escribir como un único bloque de SQL o como unión, intersección o diferencia de bloques de SQL (cada bloque de SQL consta de una sola instrucción **select-from-where**, posiblemente con cláusulas **group by** y **having**). En este apartado se estudian dos formas de componer varios bloques de SQL para expresar consultas complejas: las relaciones derivadas y la cláusula **with**.

3.8.1 Relaciones derivadas

SQL permite el uso de expresiones de subconsulta en las cláusulas **from**. Si se utiliza una expresión de este tipo, hay que proporcionar un nombre a la relación resultado y será posible renombrar los atributos mediante la cláusula **as**. Por ejemplo, considérese la subconsulta:

```
(select nombre_sucursal, avg(saldo)
  from cuenta
 group by nombre_sucursal)
 as media_sucursal (nombre_sucursal, saldo_medio)
```

Esta subconsulta genera una relación consistente en los nombres de todas las sucursales y sus correspondientes saldos de cuenta medios. El resultado de la subconsulta recibe el nombre de *media_sucursal*, con los atributos *nombre_sucursal* y *saldo_medio*.

Para ilustrar el uso de una expresión de subconsulta en la cláusula **from**, considérese la consulta “Determinar el saldo medio de las cuentas de las sucursales en las que el saldo medio de las cuentas sea superior a 1.200 €”. En el Apartado 3.5 se formulaba esta consulta utilizando la cláusula **having**. Ahora se puede reescribir sin utilizar esa cláusula, de la siguiente forma:

```
select nombre_sucursal, avg_saldo
  from (select nombre_sucursal, avg(saldo)
         from cuenta
        group by nombre_sucursal)
      as media_sucursal (nombre_sucursal, saldo_medio)
   where saldo_medio > 1200
```

Obsérvese que no hace falta utilizar la cláusula **having**, ya que la subconsulta de la cláusula **from** calcula el saldo medio y su resultado se denomina *media_sucursal*; los atributos de *media_sucursal* se pueden utilizar directamente en la cláusula **where**.

Como ejemplo adicional supóngase que se desea determinar el saldo total máximo de las sucursales. La cláusula **having** no sirve en este caso, pero esta consulta se puede escribir fácilmente usando una subconsulta en la cláusula **from** como se muestra a continuación:

```
select max(saldo_total)
  from (select nombre_sucursal, sum(saldo)
         from cuenta
        group by nombre_sucursal) as total_sucursal (nombre_sucursal, saldo_total)
```

3.8.2 La cláusula **with**

Las consultas complicadas son mucho más fáciles de formular y de entender si se estructuran descomponiéndolas en vistas más simples que luego se combinan, igual que se estructuran los programas descomponiendo sus tareas en procedimientos. Sin embargo, a diferencia de la definición de procedimientos, las cláusulas **create view** crean definiciones de vistas en la base de datos y esa definición de vista sigue en la base de datos hasta que se ejecuta una orden **drop view nombre_vista**.

La cláusula **with** proporciona una forma de definir vistas temporales cuya definición sólo está disponible para la consulta en la que aparece la cláusula. Considérese la siguiente consulta, que selecciona cuentas con el saldo máximo; si hay muchas cuentas con el mismo saldo máximo, se seleccionan todas.

```
with saldo_máximo (valor) as
    select max(saldo)
        from cuenta
select número_cuenta
from cuenta, saldo_máximo
where cuenta.saldo = saldo_máximo.valor
```

La cláusula **with**, introducida en SQL:1999 actualmente sólo está soportada en algunas bases de datos.

La consulta anterior se podría haber escrito usando una subconsulta anidada, bien en la cláusula **from**, bien en la **where**. Sin embargo, el uso de subconsultas anidadas hace que la consulta sea más difícil de leer y de entender. La cláusula **with** hace que la lógica de la consulta sea más clara; también permite utilizar definiciones de vistas en varias partes de la consulta.

Por ejemplo, supóngase que se desea determinar todas las sucursales donde el depósito total de las cuentas es mayor que la media de los depósitos totales de las cuentas de todas las sucursales. Se puede escribir la consulta usando la cláusula **with** como se muestra a continuación.

```
with total_sucursales (nombre_sucursal, valor) as
    select nombre_sucursal, sum(saldo)
        from cuenta
        group by nombre_sucursal
with media_total_sucursales(valor) as
    select avg(valor)
        from total_sucursales
select nombre_sucursal
from total_sucursales, media_total_sucursales
where total_sucursales.valor >= media_total_sucursales.valor
```

Por supuesto, se puede crear una consulta equivalente sin la cláusula **with**, pero sería más complicada y difícil de entender. Como ejercicio se puede escribir la consulta equivalente.

3.9 Vistas

Hasta este momento los ejemplos se han limitado a operar en el nivel de los modelos lógicos. Es decir, se ha dado por supuesto que las relaciones facilitadas son las relaciones reales almacenadas en la base de datos.

No resulta deseable que todos los usuarios vean el modelo lógico completo. Las consideraciones de seguridad pueden exigir que se oculten ciertos datos a los usuarios. Considérese una persona que necesita saber el número de préstamo y el nombre de la sucursal de un cliente, pero no necesita ver el importe de ese préstamo. Esa persona debería ver una relación descrita (módulo renombramiento de atributos) en SQL mediante

```
select nombre_cliente, prestatario.número_préstamo, nombre_sucursal
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo
```

Aparte de las consideraciones de seguridad, puede que se desee crear un conjunto personalizado de relaciones que se adapte mejor a la intuición de un usuario determinado que el modelo lógico. Puede que a un usuario del departamento de publicidad, por ejemplo, le guste ver una relación que consista en los clientes que tienen o bien cuenta abierta o bien préstamo concedido en el banco y las sucursales con las que trabajan. La relación que se crearía para ese empleado es la siguiente:

```
(select nombre_sucursal, nombre_cliente
from impositor, cuenta
where impositor.número_cuenta = cuenta.número_cuenta)
union
(select nombre_sucursal, nombre_cliente
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo)
```

Las relaciones que no forman parte del modelo lógico pero se hacen visibles a los usuarios como relaciones virtuales se denominan **vistas**. Es posible definir un gran número de vistas de cualquier conjunto dado de relaciones reales.

3.9.1 Definición de vistas

Las vistas en SQL se definen mediante la instrucción **create view**. Para definir una vista hay que darle un nombre e indicar la consulta que la va a calcular. La forma de la instrucción **create view** es

```
create view v as <expresión de consulta>
```

donde <expresión de consulta> es cualquier expresión legal de consulta. El nombre de la vista se representa mediante *v*.

A modo de ejemplo, considérese la vista consistente en las sucursales y sus clientes. Supóngase que se desea que esta vista se denomine *todos_los_clientes*. Esta vista se define de la manera siguiente:

```
create view todos_los_clientes as
(select nombre_sucursal, nombre_cliente
from impositor, cuenta
where impositor.número_cuenta = cuenta.número_cuenta)
union
(select nombre_sucursal, nombre_cliente
from prestatario, préstamo
where prestatario.número_préstamo = préstamo.número_préstamo)
```

Una vez definida la vista se puede utilizar su nombre para hacer referencia a la relación virtual que genera. Utilizando la vista *todos_los_clientes* se puede determinar el nombre de todos los clientes de la sucursal de Navacerrada escribiendo

```
select nombre_cliente
from todos_los_clientes
where nombre_sucursal = 'Navacerrada'
```

Los nombres de las vistas pueden aparecer en cualquier lugar en el que puedan hacerlo los nombres de las relaciones, siempre y cuando no se ejecuten sobre las vistas operaciones de actualización. El asunto de las operaciones de actualización de las vistas se estudia en el Apartado 3.10.4.

Los nombres de los atributos de las vistas pueden especificarse de manera explícita de la manera siguiente:

```
create view total_préstamos_sucursal(nombre_sucursal, total_préstamos) as
select nombre_sucursal, sum(importe)
from préstamo
group by nombre_sucursal
```

La vista anterior da la suma del importe de todos los créditos de cada sucursal. Dado que la expresión **sum(importe)** no tiene nombre, el nombre del atributo se especifica de manera explícita en la definición de la vista.

De manera intuitiva, el conjunto de tuplas de la relación de vistas es el resultado de la evaluación de la expresión de consulta que define en ese momento la vista. Por tanto, si las relaciones de vistas se calculan y se guardan, pueden quedar desfasadas si las relaciones usadas para definirlas se modifican. Para evitarlo, las vistas suelen implementarse de la manera siguiente. Cuando se define una vista, el sistema de la base de datos guarda la definición de la vista, en vez del resultado de la evaluación de la expresión del álgebra relacional que la define. Siempre que aparece una relación de vistas en una consulta, se sustituye por la expresión de consulta almacenada. Por tanto, la relación de vistas se vuelve a calcular siempre que se evalúa la consulta.

Algunos sistemas de bases de datos permiten que se guarden las relaciones de vistas, pero se aseguran de que, si las relaciones reales utilizadas en la definición de la vista cambian, la vista se mantenga actualizada. Estas vistas se denominan **vistas materializadas**. El proceso de mantener actualizada la vista se denomina **mantenimiento de vistas**, y se trata en el Apartado 14.5. Las aplicaciones en las que se utiliza frecuentemente una misma vista aprovechan las vistas materializadas, al igual que las aplicaciones que exigen una respuesta rápida a ciertas consultas basadas en las vistas. Por supuesto, las ventajas para las consultas derivadas de la materialización de las vistas deben sopesarse frente a los costes de almacenamiento y la sobrecarga añadida de las actualizaciones.

3.9.2 Vistas definidas en función de otras

En el Apartado 3.9.1 se mencionó que las relaciones de vistas pueden aparecer en cualquier lugar en que puedan hacerlo los nombres de las relaciones, salvo las restricciones al uso de las vistas en expresiones de actualización. Por tanto, se pueden utilizar vistas en las expresiones que definen otras vistas. Por ejemplo, se puede definir la vista *cliente_navacerrada* de la manera siguiente:

```
create view cliente_navacerrada as
select nombre_cliente
from todos_los_clientes
where nombre_sucursal = 'Navacerrada'
```

donde *todos_los_clientes* es, a su vez, una relación de vistas.

La **expansión de vistas** es una manera de definir el significado de las vistas definidas en términos de otras. El procedimiento asume que las definiciones de vistas no son **recursivas**; es decir, no se utiliza ninguna vista en su propia definición, ni directa ni indirectamente mediante otras definiciones de vistas. Por ejemplo, si v_1 se utiliza en la definición de v_2 , v_2 se utiliza en la definición de v_3 y v_3 se usa en la definición de v_1 , entonces v_1 , v_2 y v_3 son recursivas. Las definiciones recursivas de las vistas resultan útiles en algunos casos, y se volverá a ellas en el contexto del lenguaje Datalog, en el Apartado 5.4.

Supóngase una vista v_1 definida mediante una expresión e_1 que puede contener a su vez relaciones de vistas. Las relaciones de vistas representan a las expresiones que definen las vistas y, por tanto, se pueden sustituir por las expresiones que las definen. Si se modifica una expresión sustituyendo una relación de vistas por su definición, la expresión resultante puede seguir conteniendo otras relaciones de vistas. Por tanto, la expansión de vistas de una expresión repite la etapa de sustitución de la manera siguiente:

```
repeat
    Buscar todas las relaciones de vistas  $v_i$  de  $e_1$ 
    Sustituir la relación de vistas  $v_i$  por la expresión que define  $v_i$ 
until no quedan más relaciones de vistas en  $e_1$ 
```

Mientras las definiciones de las vistas no sean recursivas, el bucle concluirá. Por tanto, una expresión e que contenga relaciones de vistas puede entenderse como la expresión resultante de la expansión de vistas de e , que no contiene ninguna relación de vistas.

Como ilustración de la expansión de vistas considérese la expresión siguiente:

```
select *
from cliente_navacerrada
where nombre_cliente = 'Martín'
```

El procedimiento de expansión de vistas produce inicialmente

```
select *
from (select nombre_cliente
      from todos_los_clientes
      where nombre_sucursal = 'Navacerrada'
      where nombre_cliente = 'Martín'
```

luego produce

```
select *
from (select nombre_cliente
      from ((select nombre_sucursal, nombre_cliente
            from impositor, cuenta
            where impositor.número_cuenta = cuenta.número_cuenta)
            union
            (select nombre_sucursal, nombre_cliente
              from prestatario, préstamo
              where prestatario.número_préstamo = préstamo.número_préstamo))
            where nombre_sucursal = 'Navacerrada')
      where nombre_cliente = 'Martín'
```

En este momento no hay más usos de las relaciones de vistas y concluye la expansión de vistas.

3.10 Modificación de la base de datos

Hasta ahora se ha estudiado la extracción de información de las bases de datos. A continuación se mostrará cómo añadir, eliminar y modificar información utilizando SQL.

3.10.1 Borrado

Las solicitudes de borrado se expresan casi igual que las consultas. Sólo se pueden borrar tuplas completas; no se pueden borrar sólo valores de atributos concretos. SQL expresa los borrados mediante:

```
delete from r
where P
```

donde P representa un predicado y r representa una relación. La declaración **delete** busca primero todas las tuplas t en r para las que $P(t)$ es cierto y a continuación las borra de r . La cláusula **where** se puede omitir, en cuyo caso se borran todas las tuplas de r .

Obsérvese que cada comando **delete** sólo opera sobre una relación. Si se desea borrar tuplas de varias relaciones hay que utilizar una orden **delete** por cada relación. El predicado de la cláusula **where** puede ser tan complicado como la cláusula **where** de cualquier orden **select**. En el otro extremo, la cláusula **where** puede estar vacía. La consulta

```
delete from préstamo
```

borra todas las tuplas de la relación *préstamo* (los sistemas bien diseñados exigen una confirmación del usuario antes de ejecutar una petición tan devastadora).

A continuación se muestra una serie de ejemplos de peticiones de borrado en SQL:

- Borrar todas las tuplas de cuentas de la sucursal de Navacerrada.

```
delete from cuenta
where nombre_sucursal = 'Navacerrada'
```

- Borrar todos los préstamos con importe comprendido entre 1.300 € y 1.500 €.

```
delete from préstamo
where importe between 1300 and 1500
```

- Borrar todas las tuplas de cuenta de todas las sucursales de Arganzuela.

```
delete from cuenta
where nombre_sucursal in (select nombre_sucursal
from sucursal
where ciudad_sucursal = 'Arganzuela')
```

Esta solicitud de borrado busca primero todas las sucursales de Arganzuela y luego borra todas las tuplas *cuenta* correspondientes a esas sucursales.

Obsérvese que, si bien sólo se pueden borrar tuplas de una única relación a la vez, se puede hacer referencia a cualquier número de relaciones en una expresión **select-from-where** anidada en la cláusula **where** de una orden **delete**. La petición **delete** puede contener un **select** anidado que haga referencia a la relación cuyas tuplas se van a borrar. Por ejemplo, supóngase que se desea borrar los registros de todas las cuentas con saldos inferiores a la media del banco. Se puede escribir:

```
delete from cuenta
where saldo < (select avg (saldo)
from cuenta)
```

El comando **delete** comprueba primero cada tupla de la relación *cuenta* para ver si la cuenta tiene un saldo inferior a la media del banco. Luego se borran todas las tuplas que cumplen la condición; es decir, representan una cuenta con un saldo inferior a la media. Es importante realizar todas las comprobaciones antes de llevar a cabo ningún borrado; si se borra alguna tupla antes de comprobar las demás, el saldo medio puede cambiar ¡y el resultado final del borrado dependería del orden en que se procesaran las tuplas!

3.10.2 Inserción

Para insertar datos en una relación, se especifica la tupla que se desea insertar o se formula una consulta cuyo resultado sea el conjunto de tuplas que se desea insertar. Obviamente, los valores de los atributos de las tuplas que se inserten deben pertenecer al dominio de los atributos. De igual modo, las tuplas insertadas deben ser de la aridad correcta.

La instrucción **insert** más sencilla es una solicitud de inserción de una tupla. Supóngase que se desea insertar el hecho de que hay una cuenta C-9732 en la sucursal de Navacerrada y que tiene un saldo de 1.200 €. Hay que escribir

```
insert into cuenta
values ('C-9732', 'Navacerrada', 1200)
```

En este ejemplo los valores se especifican en el mismo orden en que aparecen los atributos correspondientes en el esquema de la relación. Para beneficio de los usuarios, que puede que no recuerden el orden de los atributos, SQL permite que los atributos se especifiquen en la cláusula **insert**. Por ejemplo, las siguientes instrucciones **insert** tienen una función idéntica a la anterior:

```
insert into cuenta (número_cuenta, nombre_sucursal, saldo)
values ('C-9732', 'Navacerrada', 1200)
```

```
insert into cuenta (nombre_sucursal, número_cuenta, saldo)
values ('Navacerrada', 'C-9732', 1200)
```

De forma más general es posible que se desee insertar las tuplas que resultan de una consulta. Supóngase que se les desea ofrecer, como regalo, a todos los clientes titulares de préstamos de la sucursal de Navacerrada una cuenta de ahorro con 200 € por cada préstamo que tengan concedido. Así, se escribe:

```
insert into cuenta
  select número_préstamo, nombre_sucursal, 200
  from préstamo
  where nombre_sucursal = 'Navacerrada'
```

En lugar de especificar una tupla, como se hizo en los primeros ejemplos de este apartado, se utiliza una instrucción **select** para especificar un conjunto de tuplas. SQL evalúa en primer lugar la instrucción **select**, lo que produce un conjunto de tuplas que se inserta a continuación en la relación *cuenta*. Cada tupla tiene un *número_préstamo* (que sirve como número de cuenta para la nueva cuenta), un *nombre_sucursal* (Navacerrada) y un saldo inicial de la cuenta (200 €).

También hay que añadir tuplas a la relación *impositor*; para ello hay que escribir

```
insert into impostor
  select nombre_cliente, número_préstamo
  from prestatario, préstamo
  where prestatario.número_préstamo = préstamo.número_préstamo and
    nombre_sucursal = 'Navacerrada'
```

Esta consulta inserta en la relación *impositor* una tupla (*nombre_cliente*, *número_préstamo*) por cada *nombre_cliente* que tenga concedido un préstamo en la sucursal de Navacerrada, con número de préstamo *número_préstamo*.

Es importante que la evaluación de la instrucción **select** finalice completamente antes de llevar a cabo ninguna inserción. Si se realizase alguna inserción mientras se evalúa la instrucción **select**, una petición del tipo

```
insert into cuenta
  select *
  from cuenta
```

¡Podría insertar un número infinito de tuplas! La solicitud insertaría la primera tupla de nuevo en *cuenta*, creando así una segunda copia de la tupla. Como esta segunda copia ya forma parte de *cuenta*, la instrucción **select** puede encontrarla, y se insertaría una tercera copia en *cuenta*. La instrucción **select** puede entonces encontrar esta tercera copia e insertar una cuarta copia, y así indefinidamente. Al evaluar por completo la instrucción **select** antes de realizar ninguna inserción se evita este tipo de problemas.

La discusión de la instrucción **insert** sólo ha considerado ejemplos en los que se especificaba un valor para cada atributo de las tuplas insertadas. Es posible, como se vio en el Capítulo 2, dar valores únicamente a algunos de los atributos del esquema para las tuplas insertadas. A los atributos restantes se les asigna un valor nulo, que se denota por *null*. Considérese la petición

```
insert into cuenta
  values ('C-401', null, 1200)
```

Se sabe que la cuenta C-401 tiene un saldo de 1.200 €, pero no se conoce el nombre de la sucursal. Considérese la consulta

```
select número_cuenta
  from cuenta
  where nombre_sucursal = 'Navacerrada'
```

Como el nombre de la sucursal en que se ha abierto la cuenta C-401 es desconocido, no se puede determinar si es igual a "Navacerrada".

Se puede prohibir la inserción de valores nulos en atributos concretos utilizando el LDD de SQL, que se estudia en el Apartado 3.2.

La mayor parte de los productos de bases de datos tienen utilidades especiales de "carga masiva" para insertar en las relaciones grandes conjuntos de tuplas. Estas utilidades permiten leer datos de archivos de texto con formato y pueden ejecutarse mucho más rápido que la secuencia equivalente de instrucciones **insert**.

3.10.3 Actualizaciones

En determinadas situaciones puede ser deseable modificar un valor dentro de una tupla sin cambiar *todos* los valores de la misma. Para este tipo de situaciones se puede utilizar la instrucción **update**. Al igual que ocurre con **insert** y **delete**, se pueden elegir las tuplas que se van a actualizar mediante una consulta.

Supóngase que se va a realizar el pago anual de intereses y que hay que incrementar todos los saldos en un 5 por ciento. Hay que escribir

```
update cuenta
set saldo = saldo * 1.05
```

Esta instrucción de actualización se aplica una vez a cada tupla de la relación *cuenta*.

Si sólo se paga el interés a las cuentas con un saldo de 1.000 € o superior, se puede escribir

```
update cuenta
set saldo = saldo * 1.05
where saldo >= 1000
```

En general, la cláusula **where** de la instrucción **update** puede contener cualquier constructor permitido en la cláusula **where** de la instrucción **select** (incluyendo instrucciones **select** anidadas). Al igual que ocurre con **insert** y con **delete**, un **select** anidado en una instrucción **update** puede hacer referencia a la relación que se esté actualizando. Al igual que antes, SQL primero comprueba todas las tuplas de la relación para determinar si se deben actualizar y después realiza la actualización. Por ejemplo, se puede escribir la solicitud “Pagar un interés del 5 por ciento a las cuentas cuyo saldo sea mayor que la media” como sigue:

```
update cuenta
set saldo = saldo * 1.05
where saldo > (select avg (saldo)
                from cuenta)
```

Supóngase que las cuentas con saldos superiores a 10.000 € reciben un 6 por ciento de interés, mientras que las demás reciben un 5 por ciento. Se pueden escribir dos instrucciones de actualización:

```
update cuenta
set saldo = saldo * 1.06
where saldo > 10000
```

```
update cuenta
set saldo = saldo * 1.05
where saldo <= 10000
```

Obsérvese que, como se vio en el Capítulo 2, el orden de las dos instrucciones **update** es importante. Si se cambiara el orden de las dos instrucciones, una cuenta con un saldo igual o muy poco inferior a 10.000 € recibiría un 11,3 por ciento de interés.

SQL ofrece un constructor **case** que se puede utilizar para llevar a cabo las dos instrucciones de actualización anteriores en una única instrucción **update**, evitando el problema del orden de actualización.

```
update cuenta
set saldo = case
            when saldo <= 10000 then saldo * 1.05
            else saldo * 1.06
        end
```

La forma general de la instrucción **case** es la siguiente:

```

case
  when pred1 then result1
  when pred2 then result2
  ...
  when predn then resultn
  else result0
end

```

La operación devuelve *result_i*, donde *i* es el primero de *pred₁*, *pred₂*, ..., *pred_n* que se satisface; si ninguno de ellos se satisface, la operación devuelve *result₀*. Las instrucciones **case** se pueden usar en cualquier lugar donde se espere un valor.

3.10.4 Actualización de vistas

Aunque las vistas resultan una herramienta útil para las consultas, presentan serios problemas si se expresa con ellas actualizaciones, inserciones o borrados. La dificultad radica en que las modificaciones de la base de datos expresadas en términos de vistas deben traducirse en modificaciones de las relaciones reales del modelo lógico de la base de datos.

Para ilustrar el problema, considérese un empleado que necesita ver todos los datos de préstamos de la relación *préstamo*, excepto *importe_préstamo*. Sea *sucursal_préstamo* la vista ofrecida al empleado. Esta vista se define como

```

create view sucursal_préstamo as
  select número_préstamo, nombre_sucursal
    from préstamo

```

Como se permite que el nombre de la vista aparezca en cualquier lugar en el que pueda aparecer el nombre de una relación, el empleado puede escribir

```

insert into sucursal_préstamo
  values ('P-37', 'Navacerrada')

```

Esta inserción debe representarse mediante una inserción en la relación *préstamo*, puesto que *préstamo* es la relación real a partir de la cual el sistema de bases de datos construye la vista *sucursal_préstamo*. Sin embargo, para insertar una tupla en *préstamo* hay que tener algún valor para *importe*. Hay dos enfoques razonables para tratar esta inserción:

- Rechazar la inserción y devolver al usuario un mensaje de error.
- Insertar la tupla (P-37, “Navacerrada”, *null*) en la relación *préstamo*.

Otro problema con la modificación de la base de datos mediante vistas surge con vistas como

```

create view info_préstamo as
  select nombre_cliente, importe
    from prestatario, préstamo
    where prestatario.número_préstamo = préstamo.número_préstamo

```

Esta vista muestra el importe del préstamo para cada préstamo que tenga concedido cualquier cliente del banco. Considérese la siguiente inserción mediante esta vista:

```

insert into info_préstamo
  values ('González', 1900)

```

El único método posible de inserción de tuplas en las relaciones *prestatario* y *préstamo* es insertar (“González”, *null*) en *prestatario* y (*null*, *null*, 1900) en *préstamo*. Así se obtendrían las relaciones que pueden verse en la Figura 3.3. No obstante, esta actualización no tiene el efecto deseado, ya que la relación

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
P-11	Collado Mediano	900	Fernández	P-16
P-14	Centro	1.500	Gómez	P-11
P-15	Navacerrada	1.500	Gómez	P-23
P-16	Navacerrada	1.300	López	P-15
P-17	Centro	1.000	Pérez	P-93
P-23	Moralzarzal	2.000	Santos	P-17
P-93	Becerril	500	Sotoca	P-14
<i>nulo</i>	<i>nulo</i>	1.900	Valdivieso	P-17
			González	<i>nulo</i>

Figura 3.3 Las tuplas insertadas en *préstamo* y *prestatario*.

de vistas *info_préstamo* sigue *sin* incluir la tupla (“González”, 1900). Por tanto, no hay manera de actualizar las relaciones *prestatario* y *préstamo* mediante el uso de valores nulos para obtener la actualización deseada de *info_préstamo*.

Debido a problemas como éstos no se suelen permitir las actualizaciones de las relaciones de vistas, salvo en casos concretos. Los diferentes sistemas de bases de datos especifican condiciones diferentes para permitir las actualizaciones de las relaciones de vistas y es preciso consultar el manual de cada sistema para conocer los detalles. El problema general de la modificación de las bases de datos mediante las vistas ha sido objeto de amplia investigación, y las notas bibliográficas ofrecen indicaciones de parte de esa investigación.

En general se dice que una vista de SQL es **actualizable** (es decir, se le pueden aplicar inserciones, actualizaciones y borrados) si se cumplen todas las condiciones siguientes:

- La cláusula **from** sólo tiene una relación de base de datos.
 - La cláusula **select** sólo contiene nombres de atributos de la relación y no tiene ninguna expresión, valor agregado ni especificación **distinct**.
 - Cualquier atributo que no aparezca en la cláusula **select** puede definirse como nulo.
 - La consulta no tiene cláusulas **group** ni **having**.

Con estas restricciones, las operaciones **update**, **insert** y **delete** estarían prohibidas en la vista de ejemplo *todos_los_clientes* que se había definido anteriormente.

Supóngase que se define la vista *cuenta_centro* de la manera siguiente:

```
create view cuenta_centro as
select número_cuenta, nombre_sucursal, saldo
from cuenta
where nombre_sucursal = 'Centro'
```

Esta vista es actualizable, ya que cumple las condiciones que se han fijado anteriormente.

Pese a las condiciones para la actualización, sigue existiendo el problema siguiente. Supóngase que un usuario intenta insertar la tupla ('C-999', 'Navacerrada', 1000) en la vista *cuenta_centro*. Esta tupla puede insertarse en la relación *cuenta*, pero no aparecerá en la vista *cuenta_centro*, ya que no cumple la selección impuesta por la vista.

De manera predeterminada, SQL permitiría que la actualización se llevara a cabo. Sin embargo, pueden definirse vistas con una cláusula **with check option** al final de la definición de la vista; por tanto, si una tupla insertada en la vista no cumple la condición de la cláusula **where** de la vista, el sistema de bases de datos rechaza la inserción. De manera parecida se rechazan las actualizaciones si los valores nuevos no cumplen las condiciones de la cláusula **where**.

SQL:1999 tiene un conjunto de reglas más complejo en cuanto a la posibilidad de ejecutar inserciones, actualizaciones y borrados sobre las vistas, el cual permite las actualizaciones en una clase más amplia de vistas; sin embargo, estas reglas son demasiado complejas para estudiarlas aquí.

3.10.5 Transacciones

Una **transacción** consiste en una secuencia de instrucciones de consulta o de actualización. La norma SQL especifica que una transacción comienza implícitamente cuando se ejecuta una instrucción SQL. Una de las siguientes instrucciones SQL debe finalizar la transacción:

- **Commit work** compromete la transacción actual; es decir, hace que las actualizaciones realizadas por la transacción pasen a ser permanentes en la base de datos. Una vez comprometida la transacción, se inicia de manera automática una nueva transacción.
- **Rollback work** provoca el retroceso de la transacción actual; es decir, deshace todas las actualizaciones realizadas por las instrucciones SQL de la transacción. Por tanto, el estado de la base de datos se restaura al que tenía antes de la ejecución de la primera instrucción de la transacción.

La palabra clave **work** es opcional en ambas instrucciones.

El retroceso de transacciones resulta útil si se detecta alguna condición de error durante la ejecución de la transacción. El compromiso es parecido, en cierto sentido, a guardar los cambios de un documento que se esté editando, mientras que el retroceso es como abandonar la sesión de edición sin guardar los cambios. Una vez que la transacción ha ejecutado **commit work**, sus efectos ya no se pueden deshacer con **rollback work**. El sistema de bases de datos garantiza en caso de fallo (como puede ser un error en una de las instrucciones SQL, una caída de tensión o una caída del sistema) provocar el retroceso de los efectos de la transacción si todavía no se había ejecutado **commit work**. En caso de caída de tensión o caída del sistema, el retroceso ocurre cuando el sistema se reinicia.

Por ejemplo, para transferir dinero de una cuenta a otra hay que actualizar los saldos de dos cuentas. Las dos instrucciones de actualización forman una transacción. Un error mientras una transacción ejecuta alguna de las instrucciones tiene como consecuencia que se deshagan los efectos de las instrucciones anteriores de esa transacción, por lo que la base de datos no se queda en un estado parcialmente actualizado. En el Capítulo 15 se estudian más propiedades de las transacciones.

Si un programa termina sin ejecutar ninguno de estos comandos, las actualizaciones se comprometen o se provoca su retroceso. La norma no especifica cuál de los dos cosas tiene lugar, con lo que la elección depende de la implementación. En muchas implementaciones de SQL cada instrucción de SQL se considera de manera predeterminada como una transacción en sí misma, y se compromete en cuanto se ejecuta. El compromiso automático de cada instrucción de SQL se puede desactivar si hay que ejecutar una transacción que consta de varias instrucciones SQL. La forma de desactivación del compromiso automático depende de cada implementación SQL concreta.

Una alternativa mejor, que forma parte de la norma SQL:1999 (aunque actualmente sólo está soportada por algunas implementaciones de SQL) es permitir que se encierren varias instrucciones SQL entre las palabras clave **begin atomic ... end**. Todas las instrucciones entre esas palabras clave forman así una única transacción.

3.11 Reunión de relaciones**

SQL no sólo proporciona el mecanismo básico del producto cartesiano para reunir las tuplas de las relaciones, también ofrece (en SQL-92 y versiones posteriores) otros mecanismos para reunir relaciones, como las reuniones condicionales y las reuniones naturales, así como diversas formas de reuniones externas. Estas operaciones adicionales suelen usarse como expresiones de subconsulta en la cláusula **from**.

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
P-170	Centro	3.000	Santos	P-170
P-230	Moralzarzal	4.000	Gómez	P-230
P-260	Navacerrada	1.700	López	P-155

*préstamo**prestatario***Figura 3.4** Las relaciones *préstamo* y *prestatario*.

3.11.1 Ejemplos

Se van a ilustrar las diversas operaciones de reunión mediante las relaciones *préstamo* y *prestatario* en la Figura 3.4. Se comenzará con un ejemplo sencillo de reunión interna. La Figura 3.5 muestra el resultado de la expresión

préstamo inner join prestatario on préstamo.número_préstamo = prestatario.número_préstamo

La expresión calcula la reunión zeta de las relaciones *préstamo* y *prestatario* con la condición de reunión *préstamo.número_préstamo = prestatario.número_préstamo*. Los atributos del resultado son los atributos de la relación del lado izquierdo seguidos por los de la relación del lado derecho.

Obsérvese que el atributo *número_préstamo* aparece dos veces en la figura—la primera aparición se debe a la relación *préstamo* y la segunda a *prestatario*. La norma SQL no exige que los nombres de atributo en resultados como éstos sean únicos. Se debería usar una cláusula *as* para asignar nombres únicos a los atributos de los resultados de las consultas y subconsultas.

La relación resultado de la reunión y sus atributos se renombra usando una cláusula *as*, como se ilustra a continuación:

*préstamo inner join prestatario on préstamo.número_préstamo = prestatario.número_préstamo
as pp(número_préstamo, sucursal, importe, cliente, número_préstamo_cliente)*

La segunda aparición de *número_préstamo* se ha renombrado como *número_préstamo_cliente*. El orden de los atributos en el resultado de la reunión es importante a la hora de renombrarlos.

A continuación se toma en consideración un ejemplo de la operación reunión externa por la izquierda (*left outer-join*):

préstamo left outer join prestatario on préstamo.número_préstamo = prestatario.número_préstamo

Es posible calcular la operación reunión externa por la izquierda del modo siguiente. Primero se calcula el resultado de la reunión interna como antes. Luego, para cada tupla *t* de la relación del lado izquierdo *préstamo* que no coincide con ninguna tupla de la relación del lado derecho *prestatario* en la reunión interior se añade al resultado de la reunión una tupla *r*: los atributos de la tupla *r* que se obtienen a partir de la relación del lado izquierdo se llenan con los valores de la tupla *t* y el resto de los atributos de *r* se rellena con valores nulos. La Figura 3.6 muestra la relación resultante. Las tuplas (P-170, Centro, 3.000) y (P-230, Moralzarzal, 4.000) se reúnen con las tuplas de *prestatario* y aparecen en el resultado de la reunión interna y, por tanto, en el resultado de la reunión externa por la izquierda. Por otra parte, la tupla (P-260, Navacerrada, 1.700) no coincide con ninguna tupla de *prestatario* en la reunión interna y, por eso, en el resultado de la reunión externa por la izquierda aparece la tupla (P-260, Navacerrada, 1.700, nulo, nulo).

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
P-170	Centro	3.000	Santos	P-170
P-230	Moralzarzal	4.000	Gómez	P-230

Figura 3.5 Resultado de *préstamo inner join prestatario on
préstamo.número_préstamo = prestatario.número_préstamo*.

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
P-170	Centro	3.000	Santos	P-170
P-230	Moralzarzal	4.000	Gómez	P-230
P-260	Navacerrada	1.700	<i>nulo</i>	<i>nulo</i>

Figura 3.6 Resultado de *préstamo left outer join prestatario on*
préstamo.número_préstamo = prestatario.número_préstamo.

Finalmente, se considera un ejemplo de la operación reunión natural (**natural join**).

préstamo natural inner join prestatario

Esta expresión calcula la reunión natural de las dos relaciones. El único nombre de atributo común a *préstamo* y a *prestatario* es *número_préstamo*. La Figura 3.7 muestra el resultado de la expresión. Este resultado es parecido al de la reunión interna con la condición **on** mostrada en la Figura 3.5 ya que, de hecho, tienen la misma condición de reunión. Sin embargo, el atributo *número_préstamo* sólo aparece una vez en el resultado de la reunión natural, mientras que aparece dos veces en el de la reunión con la condición **on**.

3.11.2 Tipos y condiciones de reunión

En el Apartado 3.11.1 se vieron ejemplos de las operaciones de reunión permitidas en SQL. Las operaciones de reunión toman dos relaciones y devuelven como resultado otra relación. Aunque las expresiones de reunión externa se usan normalmente en la cláusula **from**, se pueden utilizar en cualquier lugar en el que se pueda utilizar una relación.

Cada una de las variantes de las operaciones de reunión en SQL consiste en un *tipo de reunión* y una *condición de reunión*. La condición de reunión define las tuplas de las dos relaciones a casar y los atributos que se incluyen en el resultado de la reunión. El tipo de reunión define la manera de tratar las tuplas de cada relación que no casan con ninguna tupla de la otra relación (de acuerdo con la condición de reunión). La Figura 3.8 muestra algunos de los tipos y condiciones de reunión permitidos. El primer tipo de reunión es la reunión interna y los otros tres son reuniones externas. De las tres condiciones de reunión ya se han visto la reunión **natural** y la condición **on**, y se estudiará la condición **using** más adelante en este apartado.

El uso de condiciones de reunión es obligatorio en las reuniones externas, pero es opcional en las internas (si se omite, se obtiene un producto cartesiano). Sintácticamente, la palabra clave **natural** aparece delante del tipo de reunión, como se mostró anteriormente, mientras que las condiciones **on** y **using** aparecen al final de la expresión de reunión. Las palabras clave **inner** y **outer** son opcionales, ya que su forma permite deducir si la reunión es interna o externa.

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>
P-170	Centro	3.000	Santos
P-230	Moralzarzal	4.000	Gómez

Figura 3.7 Resultado de *préstamo natural inner join prestatario*.

<i>Tipos de reunión</i>	<i>Condiciones de reunión</i>
inner join left outer join right outer join full outer join	natural on <predicado> using (A₁, A₂, ..., A_n)

Figura 3.8 Tipos y condiciones de reunión.

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>
P-170	Centro	3.000	Santos
P-230	Moralzarzal	4.000	Gómez
P-155	<i>nulo</i>	<i>nulo</i>	López

Figura 3.9 Resultado de *préstamo natural right outer join prestatario*.

El significado de la condición de reunión **natural**, en cuanto a las tuplas de las relaciones a casar, es claro. La ordenación de los atributos, dentro del resultado de la reunión natural es el siguiente. Los atributos de reunión (es decir, los atributos comunes a las dos relaciones) aparecen en primer lugar, en el orden en el que aparecen en la relación del lado izquierdo. A continuación se encuentran los demás atributos no comunes de la relación del lado izquierdo y, al final, todos los atributos no comunes de la relación del lado derecho.

La reunión externa por la derecha (**right outer join**) es simétrica a la reunión externa por la izquierda (**left outer join**). Las tuplas de la relación del lado derecho que no casan con ninguna tupla de la relación del lado izquierdo se llenan con valores nulos y se añaden al resultado de la reunión externa por la derecha.

La siguiente expresión es un ejemplo de combinación de la condición de reunión natural con el tipo de reunión externa por la derecha:

préstamo natural right outer join prestatario

La Figura 3.9 muestra el resultado de esta expresión. Los atributos del resultado vienen definidos por el tipo de reunión, que es una reunión natural; por tanto, *número_préstamo* sólo aparece una vez. Las dos primeras tuplas del resultado provienen de la reunión natural interna de *préstamo* y *prestatario*. La tupla de la relación del lado derecho (López, P-155) no casa con ninguna tupla de la relación del lado izquierdo *préstamo* en la reunión interna. Así, la tupla (P-155, *nulo*, *nulo*, López) aparece en el resultado de la reunión.

La condición de reunión **using**(A_1, A_2, \dots, A_n) es parecida a la condición de reunión natural, salvo en que los atributos de reunión son los atributos A_1, A_2, \dots, A_n , en lugar de todos los atributos comunes a ambas relaciones. Los atributos A_1, A_2, \dots, A_n sólo deben ser los comunes a ambas relaciones y sólo aparecen una vez en el resultado de la unión.

La reunión externa completa (**full outer join**) es una combinación de los tipos de reunión externa por la derecha y por la izquierda. Una vez que la operación ha calculado el resultado de la reunión interna, extiende con valores nulos las tuplas de la relación del lado izquierdo que no casan con ninguna tupla de la relación del lado derecho y las añade al resultado. De manera análoga, extiende con valores nulos las tuplas de la relación del lado derecho que no casan con ninguna tupla de la relación del lado izquierdo y las añade al resultado.

Por ejemplo, la Figura 3.10 muestra el resultado de la expresión

préstamo full outer join prestatario using (número_préstamo)

Como ejemplo adicional del uso de la operación reunión externa se puede escribir la consulta: “Determinar todos los clientes que tienen una cuenta abierta pero no tienen ningún préstamo concedido en el banco” como

<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>	<i>nombre_cliente</i>
P-170	Centro	3.000	Santos
P-230	Moralzarzal	4.000	Gómez
P-260	Navacerrada	1.700	<i>nulo</i>
P-155	<i>nulo</i>	<i>nulo</i>	López

Figura 3.10 Resultado de *préstamo full outer join prestatario using (número_préstamo)*.

```
select i_NC
from (impositor left outer join prestatario
      on impostor.nombre_cliente = prestatario.nombre_cliente)
      as db1 (i_NC, número_cuenta, p_NC, número_préstamo)
where p_NC is null
```

De forma análoga se puede escribir la consulta “Determinar todos los clientes que tienen una cuenta abierta o un préstamo concedido en el banco (pero no las dos cosas)” con el operador de reunión natural externa completa como:

```
select nombre_cliente
from (impositor natural full outer join prestatario)
      where número_cuenta is null or número_préstamo is null
```

SQL-92 proporciona también otros dos tipos de reunión, denominados **cross join** (reunión cruzada) y **union join** (reunión de unión). El primero es equivalente a una reunión interna sin condición de reunión; el segundo es equivalente a una reunión externa completa con condición “false”, es decir, donde la reunión interna está vacía.

3.12 Resumen

- Los sistemas comerciales de bases de datos no utilizan la concisa y formal álgebra relacional que se trató en Capítulo 2. El ampliamente usado lenguaje SQL, que se ha estudiado en este capítulo, está basado en el álgebra relacional, pero incluye mucho “azúcar sintáctico”.
- El lenguaje de definición de datos de SQL se utiliza para crear relaciones con esquemas especificados. El LDD de SQL soporta diferentes tipos de datos, como **date** y **time**. Se pueden ver más detalles sobre el LDD de SQL, en especial sobre el soporte de las restricciones de integridad, en el Apartado 3.2.
- SQL incluye varios constructores del lenguaje de consultas a la base de datos. Todas las operaciones del álgebra relacional, incluidas las operaciones del álgebra relacional extendida, se pueden expresar en SQL. SQL también permite la ordenación de los resultados de las consultas según los atributos especificados.
- SQL trata las consultas sobre relaciones que contienen valores nulos añadiendo el valor lógico “desconocido” a los valores lógicos habituales de cierto y falso.
- SQL permite subconsultas anidadas en las cláusulas **where**. La consulta más externa puede llevar a cabo gran variedad de operaciones sobre el resultado de la subconsulta, como la comprobación de relaciones vacías o de pertenencia de valores al resultado de una subconsulta. Las subconsultas de la cláusula **from** se denominan relaciones derivadas.
- Las relaciones se definen como relaciones que contienen el resultado de consultas. Las vistas resultan útiles para ocultar información innecesaria y para recopilar información de más de una relación en una única vista.
- Las vistas temporales definidas con la cláusula **with** también resultan útiles para descomponer consultas complejas en partes más pequeñas y más fáciles de comprender.
- SQL proporciona constructores para actualizar, insertar y borrar información. Las actualizaciones mediante vistas sólo se permiten cuando se cumplen algunas condiciones bastante restrictivas.
- Las transacciones son secuencias de consultas y de actualizaciones que, en su conjunto, desempeñan una tarea. Las transacciones se pueden comprometer o retroceder; cuando se hace retroceder una transacción, se deshacen los efectos de todas las actualizaciones llevadas a cabo por esa transacción.
- SQL soporta varios tipos de reunión externa con diferentes tipos de condiciones de reunión.

*conductor (número_carné, nombre, dirección)
 coche (matrícula, modelo, año)
 accidente (número_parte, fecha, lugar)
 posee (número_carné, matrícula)
 participó (número_carné, coche, número_parte, importe_daños)*

Figura 3.11 Base de datos de seguros.

Términos de repaso

- LDD: lenguaje de definición de datos.
 - LMD: lenguaje de manipulación de datos.
 - Cláusula **select**.
 - Cláusula **from**.
 - Cláusula **where**.
 - Cláusula **as**.
 - Variable tupla.
 - Cláusula **order by**.
 - Valores duplicados.
 - Operaciones con conjuntos:
 - union, intersect, except**.
 - Funciones de agregación:
 - avg, min, max, sum, count**.
 - group by**.
 - Valores nulos.
 - Valor lógico “desconocido”.
 - Subconsultas anidadas.
 - Operaciones con conjuntos:
- {<, <=, >, >=} { **some, all** }.
 - exists**.
 - unique**.
 - Relaciones derivadas (en la cláusula **from**).
 - Cláusula **with**.
 - Vistas.
 - Definición de vistas.
 - Expansión de vistas.
 - Modificación de la base de datos.
 - delete, insert, update**.
 - Actualización de vistas.
 - Transacciones.
 - Compromiso.
 - Retroceso.
 - Tipos de reunión:
 - Reuniones interna y externa.
 - Reuniones externas por la izquierda, por la derecha y completa.
 - natural, using y on**.

Ejercicios prácticos

- 3.1 Considérese la base de datos de seguros de la Figura 3.11, en la que las claves primarias se han subrayado. Formúlense las siguientes consultas SQL para esta base de datos relacional.
- a. Determinar el número total de personas cuyos coches se hayan visto involucrados en un accidente en 2005.
 - b. Añadir un nuevo accidente a la base de datos; supóngase cualquier valor para los atributos necesarios.
 - c. Borrar el Mazda de “Martín Gómez”.
- 3.2 Considérese la base de datos de empleados de la Figura 3.12, donde las claves primarias se han subrayado. Proporcionese una expresión SQL para cada una de las consultas siguientes.
- a. Determinar el nombre y ciudad de residencia de todos los empleados que trabajan en el Banco Importante.

*empleado (nombre_empleado, calle, ciudad)
 trabaja (nombre_empleado, nombre_empresa, sueldo)
 empresa (nombre_empresa, ciudad)
 jefe (nombre_empleado, nombre_jefe)*

Figura 3.12 Base de datos de empleados.

- b. Determinar el nombre, domicilio y ciudad de residencia de todos los empleados que trabajan en el Banco Importante y ganan más de 10.000 €.
- c. Determinar todos los empleados de la base de datos que no trabajan para el Banco Importante.
- d. Determinar todos los empleados de la base de datos que ganan más que cualquier empleado del Banco Pequeño.
- e. Supóngase que las empresas pueden tener sede en varias ciudades. Determinar todas las empresas con sede en todas las ciudades en las que tiene sede el Banco Pequeño.
- f. Determinar la empresa que tiene el mayor número de empleados.
- g. Determinar las empresas cuyos empleados ganan un sueldo más alto, en media, que el sueldo medio del Banco Importante.

3.3 Considérese la base de datos relacional de la Figura 3.12. Formúlese una expresión en SQL para cada una de las siguientes consultas.

- a. Modificar la base de datos de forma que Santos viva en Tres Cantos
- b. Incrementar en un 10 por ciento el sueldo de todos los jefes del Banco Importante, a menos que su sueldo pase a ser mayor de 100.000 €, en cuyo caso se incrementará su sueldo sólo en un 3 por ciento.

3.4 SQL-92 proporciona una operación *n*-aria denominada **coalesce** (fusión) que se define del modo siguiente: **coalesce**(A_1, A_2, \dots, A_n) devuelve el primer A_i no nulo de la lista A_1, A_2, \dots, A_n y devuelve un valor nulo si todos son nulos.

Sean a y b relaciones con los esquemas $A(\text{nombre}, \text{dirección}, \text{puesto})$ y $B(\text{nombre}, \text{dirección}, \text{sueldo})$, respectivamente. Indíquese la manera de expresar a **natural full outer join** b , utilizando la operación **full outer-join** con una condición **on** y la operación **coalesce**. Compruébese que la relación resultado no contiene dos copias de los atributos *nombre* y *dirección* y que la solución es válida aunque alguna tupla de a o de b tenga valores nulos para los atributos *nombre* o *dirección*.

3.5 Supóngase que se tiene una relación $\text{notas}(id_estudiante, puntuación)$ y que se quiere clasificar a los estudiantes en función de la puntuación del modo siguiente: *SS* si $puntuación < 40$, *AP* si $40 \leq puntuación < 60$, *NT* si $60 \leq puntuación < 80$ y *SB* si $80 \leq puntuación$. Escribanse consultas para hacer lo siguiente:

- a. Mostrar la clasificación de cada estudiante, en términos de la relación *notas*.
- b. Determinar el número de estudiantes en cada clase.

3.6 Considérese la consulta SQL

```
select p.a1
  from p, r1, r2
 where p.a1 = r1.a1 or p.a1 = r2.a1
```

¿Bajo qué condiciones la consulta anterior selecciona los valores de $p.a1$ que están exclusivamente en $r1$ o en $r2$? Examínense cuidadosamente los casos en los que $r1$ o $r2$ pueden estar vacíos.

3.7 Algunos sistemas permiten los valores nulos *marcados*. Un valor nulo marcado \perp_i es igual a sí mismo, pero si $i \neq j$, entonces $\perp_i \neq \perp_j$. Una aplicación de los valores nulos marcados es permitir ciertas actualizaciones mediante vistas. Considérese la vista *info_préstamo* (Apartado 3.9). Muéstrela la manera en que se pueden utilizar los valores nulos marcados para permitir la inserción de la tupla (“González”, 1900) mediante *info_préstamo*.

Ejercicios

3.8 Considérese la base de datos de seguros de la Figura 3.11, en la que las claves primarias están subrayadas. Créense las siguientes consultas SQL para esta base de datos relacional.

- a. Determinar el número de accidentes en que han estado implicados los coches pertenecientes a “Martín Gómez”.

- b. Actualizar a 3.000 € el importe de los daños del coche con número de matrícula “2000 ABC” en el accidente con número de parte “PA2197”.
- 3.9 Considérese la base de datos de empleados de la Figura 3.12, en la que las claves primarias se han subrayado. Proporcióñese una expresión SQL para cada una de las consultas siguientes.
- Determinar el nombre de todos los empleados que trabajan en el Banco Importante.
 - Determinar todos los empleados de la base de datos que viven en la misma ciudad que la empresa para la que trabajan.
 - Determinar todos los empleados de la base de datos que viven en la misma ciudad y en la misma calle que sus jefes.
 - Determinar todos los empleados que ganan más que el sueldo medio de los empleados de su empresa.
 - Determinar la empresa que tiene la nómina más pequeña.
- 3.10 Considérese la base de datos relacional de la Figura 3.12. Formúlese una expresión en SQL para cada una de las siguientes consultas:
- Incrementar en un 10 por ciento el sueldo de todos los empleados del Banco Importante.
 - Incrementar en un 10 por ciento el sueldo de todos los jefes del Banco Importante.
 - Borrar todas las tuplas de la relación *trabaja* correspondientes a los empleados del Banco Importante.
- 3.11 Considérense los esquemas de relación siguientes:
- $$R = (A, B, C)$$
- $$S = (D, E, F)$$
- Considérense también las relaciones $r(R)$ y $s(S)$. Obténgase la expresión SQL equivalente a las siguientes consultas:
- $\Pi_A(r)$
 - $\sigma_{B=17}(r)$
 - $r \times s$
 - $\Pi_{A,F}(\sigma_{C=D}(r \times s))$
- 3.12 Sea $R = (A, B, C)$ y sean r_1 y r_2 relaciones sobre el esquema R . Proporcióñese una expresión SQL equivalente a cada una de las siguientes consultas:
- $r_1 \cup r_2$
 - $r_1 \cap r_2$
 - $r_1 - r_2$
 - $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$
- 3.13 Demuéstrese que en SQL $<> \text{all}$ es equivalente a **not in**.
- 3.14 Considérese la base de datos relacional de la Figura 3.12. Utilizando SQL, defínase una vista consistente en *nombre_jefe* y el sueldo medio de todos los empleados que trabajan para ese jefe. Explíquese por qué el sistema de base de datos no debería permitir que las actualizaciones se expresaran en términos de esta vista.
- 3.15 Escríbase una consulta SQL, sin usar la cláusula **with**, para determinar todas las sucursales en las que el saldo total de las cuentas sea menor que la media del saldo total de todas las sucursales,
- Usando una consulta anidada en la cláusula **from**.
 - Usando una consulta anidada en una cláusula **having**.
- 3.16 Determínense dos razones por las que se puedan introducir valores nulos en una base de datos.
- 3.17 Mostrar la manera de expresar la operación **coalesce** del Ejercicio 3.4 mediante la operación **case**.
- 3.18 Dada una definición de esquema SQL para la base de datos de empleados de la Figura 3.12, elijase un dominio adecuado para cada atributo y una clave primaria adecuada para cada esquema de relación.

3.19 Usando las relaciones de la base de datos bancaria de ejemplo, escríbanse expresiones SQL para definir las vistas siguientes:

- Una vista que contenga el número de cuenta y el nombre de los clientes (pero no el saldo) de todas las cuentas de la sucursal de El Escorial.
- Una vista que contenga el nombre y dirección de todos los clientes que tengan una cuenta abierta en el banco, pero que no tengan concedido ningún préstamo.
- Una vista que contenga el nombre y saldo medio de la cuenta de todos los clientes de la sucursal de Collado Villalba.

3.20 Para cada una de las vistas definidas en el Ejercicio 3.19, explíquese cómo se llevarían a cabo las actualizaciones (si es que se deben permitir).

3.21 Considérese el siguiente esquema relacional

$$\begin{aligned} &\text{empleado}(\underline{\text{número_empleado}}, \text{nombre}, \text{sucursal}, \text{edad}) \\ &\text{libros}(\underline{\text{isbn}}, \text{título}, \text{autores}, \text{editorial}) \\ &\text{préstamo}(\underline{\text{número_empleado}}, \underline{\text{isbn}}, \text{fecha}) \end{aligned}$$

Escríbanse las siguientes consultas en SQL.

- Escribir el nombre de los empleados que hayan pedido algún libro publicado por McGraw-Hill.
- Escribir el nombre de los empleados que hayan pedido todos los libros publicados por McGraw-Hill.
- Para cada editorial, escribir el nombre de los empleados que han pedido más de cinco libros de esa editorial.

3.22 Considérese el esquema relacional

$$\begin{aligned} &\text{estudiante}(\underline{\text{id_estudiante}}, \text{nombre_estudiante}) \\ &\text{matriculado}(\underline{\text{id_estudiante}}, \underline{\text{id_asignatura}}) \end{aligned}$$

Escríbase una consulta SQL para obtener el ID de estudiante y el nombre de cada estudiante, así como el número total de asignaturas en las que se ha matriculado. Los estudiantes que no estén matriculados en ninguna asignatura deben aparecer también, con el número de asignaturas en las que se han matriculado como 0.

3.23 Supóngase que se tiene una relación *notas*(*id_estudiante*, *puntuación*). Escríbase una consulta SQL para determinar la *clasificación densa* de cada estudiante. Es decir, todos los estudiantes con la nota más alta obtienen una clasificación de 1, los que tienen la siguiente nota más alta obtienen una clasificación de 2, etc. *Sugerencia:* divídase la tarea en partes con la cláusula **with**.

Notas bibliográficas

La versión original de SQL, denominada Sequel 2, se describe en Chamberlin et al. [1976]. Sequel 2 procede de los lenguajes Square (Boyce et al. [1975] y Chamberlin y Boyce [1974]). La norma American National Standard SQL-86 se describe en ANSI [1986]. La definición de SQL según la Arquitectura de aplicación de sistemas (System Application Architecture) de IBM se describe en IBM [1987]. Las normas oficiales de SQL-89 y de SQL-92 están disponibles en ANSI [1989] y ANSI [1992], respectivamente.

Entre las descripciones del lenguaje SQL-92 en libros de texto están Date y Darwen [1997], Melton y Simon [1993] y Cannan y Otten [1993]. Date y Darwen [1997] y Date [1993a] incluyen una crítica de SQL-92.

Entre los libros de texto sobre SQL:1999 están Melton y Simon [2001] y Melton [2002]. Eisenberg y Melton [1999] ofrecen una visión general de SQL:1999. Donahoo y Speegle [2005] abordan SQL desde el punto de vista de los desarrolladores. Eisenberg et al. [2004] ofrecen una visión general de SQL:2003.

Las normas SQL:1999 y SQL:2003 están publicadas como conjuntos de documentos de normas ISO/IEC, que se describen con más detalle en el Apartado 23.3. Los documentos de normas tienen gran den-

sidad de información, resultan difíciles de leer y son útiles, sobre todo, para los implementadores de bases de datos. Los documentos de normas están disponibles para su compra electrónica en el sitio web <http://webstore.ansi.org>.

Muchos productos de bases de datos soportan más características de SQL que las especificadas en las normas, y puede que no soporten algunas características de la norma. Se puede encontrar más información sobre estas características en los manuales de usuario de SQL de los productos respectivos.

El procesamiento de las consultas SQL, incluidos los algoritmos y las consideraciones sobre rendimiento, se estudia en los Capítulos 13 y 14. En esos mismos capítulos hay referencias bibliográficas al respecto.

Las reglas usadas por SQL para determinar si es posible actualizar una vista y la manera en que se reflejan estas actualizaciones en las relaciones subyacentes de las bases de datos se definen en la norma SQL:1999 y están resumidas en Melton y Simon [2001].

SQL avanzado

En el Capítulo 3 se ofreció un tratamiento detallado de la estructura básica de SQL. El lenguaje SQL ha evolucionado a partir de finales de los años setenta desde un lenguaje con pocas funcionalidades a un lenguaje complejo con características para satisfacer a muchos tipos diferentes de usuarios. En este capítulo se tratarán algunas de las características avanzadas de SQL. Se seguirá empleando en los ejemplos el esquema bancario, que se reproduce por comodidad en la Figura 4.1.

4.1 Tipos de datos y esquemas

Ya hemos visto que debe haber asociado un tipo, es decir, un dominio de valores posibles, con cada atributo. En el Capítulo 3 se vieron varios tipos de datos predefinidos soportados por SQL, como los tipos enteros, los reales y los de carácter. Hay otros tipos de datos predefinidos soportados por SQL, que se describirán a continuación. También se describirá la manera de crear en SQL tipos sencillos definidos por los usuarios.

4.1.1 Tipos de datos predefinidos

Además de los tipos de datos básicos que se presentaron en el Apartado 3.2, la norma SQL soporta otros tipos de datos predefinidos, como:

- **date**. Una fecha de calendario que contiene el año (de cuatro cifras), el mes y el día del mes.
- **time**. La hora del día, en horas, minutos y segundos. Se puede usar una variante, **time(*p*)**, para especificar el número de cifras decimales para los segundos (el valor predeterminado es 0). También es posible almacenar la información del huso horario junto a la hora especificando **time with timezone**.
- **timestamp**. Una combinación de **date** y **time**. Se puede usar una variante, **timestamp(*p*)**, para especificar el número de cifras decimales para los segundos (el valor predeterminado es seis). También se almacena información sobre el huso horario si se especifica **with timezone**.

```

sucursal (nombre_sucursal, ciudad_sucursal, activos)
cliente (nombre_cliente, calle_cliente, ciudad_cliente)
préstamo (número_préstamo, nombre_sucursal, importe)
prestatario (nombre_cliente, número_préstamo)
cuenta (número_cuenta, nombre_sucursal, saldo)
impositor (nombre_cliente, número_cuenta)

```

Figura 4.1 Esquema de la entidad bancaria.

Los valores de fecha y hora se pueden especificar de esta manera:

```
date '2001-04-25'
time '09:30:00'
timestamp '2001-04-25 10:29:01.45'
```

La fecha se debe especificar en el formato de año seguido del mes y del día, tal y como se muestra. El campo segundos de **time** y **timestamp** puede tener parte decimal, como puede verse en el ejemplo anterior.

Se pueden usar expresiones de la forma **cast e as t** para convertir una cadena de caracteres (o una expresión de tipo cadena de caracteres) *e* al tipo *t*, donde *t* es de tipo **date**, **time** o **timestamp**. La cadena de caracteres debe tener el formato adecuado, como se indicó al comienzo de este párrafo. Si es necesaria, la información sobre el huso horario puede deducirse de la configuración del sistema.

Para extraer campos concretos de un valor *d* de **date** o de **time** se puede utilizar **extract (campo from d)**, donde *campo* puede ser **year**, **month**, **day**, **hour**, **minute** o **second**. La información sobre el huso horario puede obtenerse mediante **timezone_hour** y **timezone_minute**.

SQL también define varias funciones útiles para obtener la fecha y la hora actuales. Por ejemplo, **current_date** devuelve la fecha actual, **current_time** devuelve la hora actual (con su huso horario) y **localtime** devuelve la hora local actual (sin huso horario). Las marcas de tiempo (fecha y hora) se obtienen con **current_timestamp** (con huso horario) y **localtimestamp** (fecha y hora locales sin huso horario).

SQL permite realizar operaciones de comparación sobre todos los tipos de datos que se han mencionado aquí, así como operaciones aritméticas y de comparación sobre los diferentes tipos de datos numéricos. SQL también proporciona un tipo de datos denominado **interval** y permite realizar cálculos basados en fechas, horas e intervalos. Por ejemplo, si *x* e *y* son del tipo **date**, entonces *x - y* es un intervalo cuyo valor es el número de días desde la fecha *x* hasta la *y*. De forma análoga, al sumar o restar un intervalo a una fecha o a una hora se obtiene como resultado otra fecha u hora, respectivamente.

A menudo resulta útil comparar valores de diferentes tipos de datos **compatibles**. Por ejemplo, supóngase que el tipo de datos de *nombre_cliente* es una cadena de caracteres de longitud 20 y el tipo de datos de *nombre_sucursal* es una cadena de caracteres de longitud 15. Aunque la longitud de las cadenas sea diferente, la norma SQL considera que los dos tipos de datos son compatibles. Como ejemplo adicional, como cada entero perteneciente al tipo **smallint** es también un entero, las comparaciones *x < y*, donde *x* es de tipo **smallint** e *y* es de tipo **int** (o viceversa), son válidas. Este tipo de comparación se lleva a cabo transformando primero el número *x* al tipo **int**. Las transformaciones de este tipo se denominan **coerción**. La **coerción de tipos** se emplea de manera habitual en los lenguajes de programación comunes, así como en los sistemas de bases de datos.

4.1.2 Tipos definidos por los usuarios

SQL soporta dos formas de tipos de datos definidos por los usuarios. La primera forma, que se tratará a continuación, se denomina **alias de tipos (distinct types)**. La otra forma, denominada **tipos de datos estructurados**, permite la creación de tipos de datos complejos, que pueden anidar estructuras de registro, arrays y multiconjuntos. En este capítulo no se tratan los tipos de datos complejos, pero se describen más adelante, en el Capítulo 9.

Varios atributos pueden ser del mismo tipo de datos. Por ejemplo, los atributos *nombre_cliente* y *nombre_empleado* pueden tener el mismo dominio: el conjunto de todos los nombres propios. No obstante, los dominios de *saldo* y *nombre_sucursal*, ciertamente, deben ser diferentes. Quizás resulte menos evidente si *nombre_cliente* y *nombre_sucursal* deben tener el mismo dominio. En el nivel de implementación tanto los nombres de los clientes como los de las sucursales son cadenas de caracteres. Sin embargo, normalmente no se considera que la consulta “Determinar todos los clientes que tengan el mismo nombre que alguna sucursal” sea una consulta con sentido. Por tanto, si se considera la base de datos desde el nivel conceptual, en vez de hacerlo desde el nivel físico, *nombre_cliente* y *nombre_sucursal* deben tener dominios distintos.

A nivel práctico resulta más grave asignar el nombre de un cliente a una sucursal; de manera parecida, comparar directamente un valor monetario expresado en euros con otro valor monetario expresado en libras también es, seguramente, un error de programación. Un buen sistema de tipos de datos debe

poder detectar este tipo de asignaciones o comparaciones. Para soportar estas comprobaciones, SQL aporta el concepto de **alias de tipos**.

La cláusula **create type** puede utilizarse para definir tipos de datos nuevos. Por ejemplo, las instrucciones:

```
create type Euros as numeric(12,2) final
create type Libras as numeric(12,2) final
```

declaran los tipos de datos definidos por los usuarios *Euros* y *Libras* como números decimales con un total de doce cifras, dos de las cuales se hallan tras la coma decimal. (La palabra clave **final** no resulta realmente significativa en este contexto, pero la norma de SQL:1999 la exige por motivos que no vienen al caso; algunas implementaciones permiten omitir la palabra clave **final**). Los tipos recién creados pueden utilizarse, por ejemplo, como tipos de los atributos de las relaciones. Por ejemplo, se puede declarar la tabla *cuenta* como:

```
create table cuenta
(número_cuenta char(10),
nombre_sucursal char(15),
saldo Euros)
```

Cualquier intento de asignar un valor de tipo *Euros* a una variable de tipo *Libras* daría como resultado un error de compilación, aunque ambos sean del mismo tipo numérico. Una asignación de ese tipo probablemente se deba a un error de programación, en el que el programador haya olvidado las diferencias de divisas. La declaración de tipos distintos para divisas diferentes ayuda a detectar esos errores.

Como consecuencia del control riguroso de los tipos de datos, la expresión (*saldo.cuenta* + 20) no se aceptaría, ya que el atributo y la constante entera 20 tienen diferentes tipos de datos. Los valores de un tipo de datos pueden *convertirse* (*cast*) a otro dominio como se muestra a continuación:

```
cast (saldo.cuenta to numeric(12,2))
```

Se podría realizar la suma sobre el tipo *numeric*, pero para volver a guardar el resultado en un atributo del tipo *Euros* habría que emplear otra expresión *cast* para volver a convertir el tipo de datos a *Euros*.

SQL también ofrece las cláusulas **drop type** y **alter type** para eliminar o modificar los tipos de datos que se han creado anteriormente.

Antes incluso de la incorporación a SQL de los tipos de datos definidos por los usuarios (en SQL:1999), SQL tenía el concepto parecido, pero sutilmente diferente, de **tipo de dominio (domain type)** (introducido en SQL-92). Se podría definir el tipo de dominio *EEuros* de la manera siguiente:

```
create domain EEuros as numeric(12,2)
```

El tipo de dominio *EEuros* puede utilizarse como tipo de atributo, igual que se ha utilizado el tipo de datos *Euros*. Sin embargo, hay dos diferencias significativas entre los tipos de datos y los dominios:

1. Sobre los dominios se pueden especificar restricciones, como **not null**, y valores predeterminados para las variables del tipo de dominio, mientras que no se pueden especificar restricciones ni valores predeterminados sobre los tipos de datos definidos por los usuarios. Los tipos de datos definidos por los usuarios están diseñados no sólo para utilizarlos para especificar los tipos de datos de los atributos, sino también en extensiones procedimentales de SQL en las que puede que no sea posible aplicar restricciones. Más adelante se volverá al problema de las restricciones sobre los dominios, en el Apartado 4.2.4.
2. Los dominios no tienen tipos estrictos. En consecuencia, los valores de un tipo de dominio pueden asignarse a los de otro tipo de dominio, siempre y cuando los tipos subyacentes sean compatibles.

4.1.3 Tipos de datos para objetos grandes

Muchas aplicaciones de bases de datos de última generación necesitan almacenar atributos que pueden ser de gran tamaño (del orden de muchos kilobytes), como pueden ser fotografías de personas, o muy grande (del orden de muchos megabytes o, incluso, gigabytes), como las imágenes médicas de alta resolución o fragmentos de vídeo. SQL, por tanto, ofrece nuevos tipos de datos para objetos de gran tamaño para los datos de caracteres (**clob**) y para los datos binarios (**blob**). Las letras “lob” de estos tipos de datos significan “objeto grande” (Large Object). Por ejemplo, se pueden declarar los atributos

```
revista_literaria clob(10KB)
imagen blob(10MB)
película blob(2GB)
```

La ejecución de consultas SQL suele recuperar una o más filas del resultado en la memoria. Los objetos grandes suelen utilizarse en aplicaciones externas, y para objetos de tamaño muy grande (de varios megabytes a gigabytes), resulta poco eficiente o poco práctico recuperar en la memoria objetos de gran tamaño de tipo entero. En vez de eso, las aplicaciones suelen utilizar las consultas SQL para recuperar “localizadores” de los objetos de gran tamaño y luego emplean el localizador para manipular el objeto desde el lenguaje anfitrión. Por ejemplo, la interfaz de programación de aplicaciones JDBC (que se describe en el Apartado 4.5.2) permite recuperar un localizador en lugar de todo el objeto de gran tamaño; luego se puede emplear el localizador para recuperar este objeto en fragmentos más pequeños, en vez de recuperarlo todo de golpe, de manera parecida a como se leen los datos de los archivos del sistema operativo mediante llamadas a funciones de lectura.

4.1.4 Esquemas, catálogos y entornos

Para comprender la razón de que existan esquemas y catálogos, considérese la manera en que se dan nombres a los archivos en los sistemas de archivos. Los primeros sistemas de archivos eran “planos”, es decir, todos los archivos se almacenaban en un único directorio. Los sistemas de archivos de última generación, evidentemente, tienen una estructura de directorios y los archivos se almacenan en diferentes subdirectorios. Para dar un nombre único a un archivo hay que especificar su nombre completo, por ejemplo, /usuarios/avi/libro-bd/capítulo4.tex.

Al igual que los primeros sistemas de archivos, los primeros sistemas de bases de datos tenían también un único espacio de nombres para todas las relaciones. Los usuarios tenían que coordinarse para asegurarse de que no intentaban emplear el mismo nombre para relaciones diferentes. Los sistemas de bases de datos contemporáneos ofrecen una jerarquía de tres niveles para los nombres de las relaciones. El nivel superior de la jerarquía consta de **catálogos**, cada uno de los cuales puede contener **esquemas**. Los objetos de SQL, como las relaciones y las vistas, están contenidos en **esquemas**. (Algunas implementaciones de las bases de datos emplean el término “base de datos” en lugar de catálogo).

Para llevar a cabo cualquier acción sobre la base de datos los usuarios (o los programas) primero deben *conectarse* a ella. El usuario debe proporcionar el nombre de usuario y, generalmente, una contraseña secreta para que se compruebe su identidad. Cada usuario tiene un catálogo y un esquema predeterminados, y esa combinación es única para cada usuario. Cuando un usuario se conecta a un sistema de bases de datos, se configuran para la conexión el catálogo y el esquema predeterminados; esto equivale a la conversión del directorio actual en el directorio de inicio del usuario cuando éste inicia sesión en un sistema operativo.

Para identificar de manera única cada relación hay que utilizar un nombre con tres partes. Por ejemplo,

```
catálogo5.esquema_bancario.cuenta
```

Se puede omitir el componente del catálogo, en cuyo caso la parte del nombre correspondiente al catálogo se considera que es el catálogo predeterminado de la conexión. Por tanto, si **catálogo5** es el catálogo predeterminado, se puede utilizar **esquema_bancario.cuenta** para identificar de manera única la misma relación. Además, también se puede omitir el nombre del esquema y la parte correspondiente al esquema del nombre vuelve a considerarse el esquema predeterminado de la conexión. Así pues, se puede

emplear únicamente cuenta si el catálogo predeterminado es **catálogo5** y el esquema predeterminado es **esquema_bancario**.

Cuando se dispone de varios catálogos y de varios esquemas, diferentes aplicaciones y usuarios pueden trabajar de manera independiente sin preocuparse de posibles coincidencias de nombres. Además, pueden ejecutarse varias versiones de la misma aplicación—una versión de producción, otras versiones de prueba—sobre el mismo sistema de bases de datos.

El catálogo y el esquema predeterminados forman parte de un **entorno SQL** que se configura para cada conexión. El entorno contiene, además, el identificador del usuario (también denominado *identificador de autorización*). Todas las instrucciones habituales de SQL, incluidas las instrucciones de LDD y de LMD, operan en el contexto de un esquema. Se pueden crear y descartar esquemas mediante las instrucciones **create schema** y **drop schema**. La creación y la eliminación de catálogos depende de cada implementación y no forma parte de la norma de SQL.

4.2 Restricciones de integridad

Las restricciones de integridad garantizan que las modificaciones realizadas en la base de datos por los usuarios autorizados no den lugar a una pérdida de la consistencia de los datos. Por tanto, las restricciones de integridad protegen contra daños accidentales a las bases de datos.

Algunos ejemplos de restricciones de integridad son:

- El saldo de las cuentas no puede ser nulo.
- No puede haber dos cuentas con el mismo número.
- Todos los números de cuenta de la relación *impositor* deben tener el número de cuenta correspondiente en la relación *cuenta*.
- El salario por hora de los empleados del banco debe ser, como mínimo, de 6,00 € la hora.

En general, las restricciones de integridad pueden ser predicados arbitrarios que hagan referencia a la base de datos. Sin embargo, puede resultar costosa la comprobación de estos predicados arbitrarios. Por tanto, la mayor parte de los sistemas de bases de datos permiten especificar restricciones de integridad que puedan probarse con una sobrecarga mínima. En el Capítulo 7 se estudia otra forma de restricción de integridad, denominada **dependencia funcional**, que se utiliza sobre todo en el proceso de diseño de esquemas.

4.2.1 Restricciones sobre una sola relación

En el Apartado 3.2 se describió la manera de definir tablas mediante el comando **create table**. Este comando también puede incluir instrucciones para restricciones de integridad. Además de la restricción “de clave primaria”, hay varias más que pueden incluirse en el comando **create table**. Entre las restricciones de integridad permitidas se encuentran:

- **not null**
- **unique**
- **check(<predicado>)**

Cada uno de estos tipos de restricciones se trata en los apartados siguientes.

4.2.2 Restricción **not null**

Como se estudió en el Capítulo 2, el valor nulo (*null*) es miembro de todos los dominios y, en consecuencia, de manera predeterminada es un valor legal para todos los atributos de SQL. Para determinados atributos, sin embargo, los valores nulos pueden resultar poco adecuados. Considérese una tupla de la relación *cuenta* en la que *número_cuenta* sea nulo. Este tipo de tupla proporciona información de las cuentas desconocidas; por tanto, no contiene información útil. De manera parecida, no es deseable que

el saldo de la cuenta sea un valor nulo. En estos casos así se deseará prohibir los valores nulos, lo que se puede hacer restringiendo el dominio de los atributos *número_cuenta* y *saldo* para excluir los valores nulos, declarándolos de la manera siguiente:

```
número_cuenta char(10) not null
saldo numeric(12,2) not null
```

La especificación **not null** prohíbe la inserción de valores nulos para ese atributo. Cualquier modificación de la base de datos que haga que se inserte un valor nulo en un atributo declarado como **not null** genera un diagnóstico de error.

Existen muchas situaciones en las que se desea evitar los valores nulos. En concreto, SQL prohíbe los valores nulos en la clave primaria de los esquemas de las relaciones. Por tanto, en el ejemplo bancario, en la relación *cuenta*, si el atributo *número_cuenta* se declara clave primaria de *cuenta*, no puede adoptar valores nulos. En consecuencia, no hace falta declararlo **not null** de manera explícita.

La especificación **not null** también puede aplicarse a declaraciones de dominio definidas por los usuarios; en consecuencia, no se permitirá a los atributos de ese tipo de dominio tomar valores nulos. Por ejemplo, si se desea que el dominio *Euros* no tome valores nulos, se puede declarar de la manera siguiente:

```
create domain Euros numeric(12,2) not null
```

4.2.3 Restricción unique

SQL también soporta la restricción de integridad

$$\text{unique } (A_{j_1}, A_{j_2}, \dots, A_{j_m})$$

La especificación **unique** indica que los atributos $A_{j_1}, A_{j_2}, \dots, A_{j_m}$ forman una clave candidata; es decir, ningún par de tuplas de la relación puede ser igual en todos los atributos de la clave primaria. Sin embargo, se permite que los atributos de la clave candidata tengan valores nulos, a menos que se hayan declarado de manera explícita como **not null**. Recuérdese que los valores nulos no son iguales a ningún otro valor. (El tratamiento de los valores nulos en este caso es el mismo que el del constructor **unique** definido en el Apartado 3.7.4).

4.2.4 La cláusula check

La cláusula **check** de SQL puede aplicarse a declaraciones de relaciones y a declaraciones de dominios. Cuando se aplica a declaraciones de relaciones, la cláusula **check(*P*)** especifica un predicado *P* que deben cumplir todas las tuplas de la relación.

Un uso frecuente de la cláusula **check** es garantizar que los valores de los atributos cumplan las condiciones especificadas, lo que permite en realidad construir un potente sistema de tipos. Por ejemplo, la cláusula **check(*activos*>=0)** en el comando **create table** de la relación *sucursal* garantiza que el valor de **activos** no sea negativo.

Considérese ahora el siguiente ejemplo:

```
create table estudiante
  (nombre          char(15) not null,
   id_estudiante  char(10),
   tipo_titulación char(15),
   primary key (id_estudiante),
   check (tipo_titulación in ('Diplomatura', 'Licenciatura', 'Doctorado')))
```

En este caso se utiliza la cláusula **check** para simular un tipo enumerado, especificando que *tipo_titulación* debe ser 'Diplomatura', 'Licenciatura' o 'Doctorado'.

Cuando se aplica a un dominio, la cláusula **check** permite a los diseñadores de esquemas especificar un predicado que debe cumplir cualquier valor asignado a las variables cuyo tipo de datos sea el dominio.

Por ejemplo, una cláusula **check** puede garantizar que el dominio del salario por hora sólo permita valores mayores que un mínimo especificado (como puede ser el salario mínimo):

```
create domain SalarioPorHora numeric(5,2)
constraint prueba_valor_salario check(value >= 6.00)
```

Se ha impuesto sobre el dominio *SalarioPorHora* una restricción que garantiza que el salario por hora sea mayor o igual que 6.00. La cláusula **constraint** *prueba_valor_salario* es optativa, y se emplea para dar el nombre *prueba_valor_salario* a la restricción. El nombre lo utiliza el sistema para indicar la restricción que ha violado alguna actualización.

Como ejemplo adicional, se puede restringir un dominio para que sólo contenga un conjunto determinado de valores mediante la cláusula **in**:

```
create domain TipoCuenta char(10)
constraint prueba_tipo_cuenta
check (value in ('Corriente', 'Ahorro'))
```

Por tanto, la cláusula **check** permite restringir los atributos y los dominios de una forma potente que la mayor parte de los sistemas del tipos de los lenguajes de programación no permiten.

Las anteriores condiciones de **check** pueden comprobarse con relativa facilidad, al insertar o modificar una tupla. Sin embargo, en general, las condiciones de **check** pueden ser más complejas (y más difíciles de comprobar), ya que se permiten subconsultas que hagan referencias a otras relaciones en la condición **check**. Por ejemplo, se podría especificar esta restricción sobre la relación *cuenta*:

```
check (nombre_sucursal in (select nombre_sucursal from sucursal))
```

La condición de **check** comprueba que el *nombre_sucursal* de cada tupla de la relación *cuenta* es realmente el nombre de una sucursal de la relación *sucursal*. Por tanto, no sólo hay que comprobar la condición cuando se inserta o modifica una tupla de *cuenta*, sino también cuando se modifica la relación *sucursal* (en ese caso, cuando se borra o modifica alguna tupla de la relación *sucursal*).

La restricción anterior es, en realidad, un ejemplo de una clase de constantes denominadas restricciones de *integridad referencial*. Este tipo de restricciones, junto con una manera más sencilla de especificarlas en SQL, se estudian en el Apartado 4.2.5.

Las condiciones **check** complejas pueden ser útiles cuando se desea garantizar la integridad de los datos, pero deben emplearse con precaución, ya que pueden resultar costosas de comprobar.

4.2.5 Integridad referencial

A menudo se desea garantizar que el valor que aparece en una relación para un conjunto dado de atributos aparezca también para un conjunto determinado de atributos en otra relación. Esta condición se denomina **integridad referencial**.

Las claves externas pueden especificarse como parte de la instrucción de SQL **create table** mediante la cláusula **foreign key**. Las declaraciones de claves externas se ilustran mediante la definición en el LDD de SQL de parte de la base de datos bancaria, como puede verse en la Figura 4.2. La declaración de la tabla *cuenta* tiene una declaración “**foreign key** (*nombre_sucursal*) **references** *sucursal*”. Esta declaración de clave externa especifica que para cada tupla de *cuenta*, el nombre de sucursal especificado en la tupla debe existir en la relación *sucursal*. Sin esta restricción, es posible que alguna cuenta especifique el nombre de una sucursal inexistente.

De manera más general, sean $r_1(R_1)$ y $r_2(R_2)$ relaciones con las claves primarias C_1 y C_2 , respectivamente (recuérdese que R_1 y R_2 denotan el conjunto de atributos de r_1 y de r_2 , respectivamente). Se dice que un subconjunto α de R_2 es una **clave externa** que hace referencia a C_1 de la relación r_1 si se exige que, para cada tupla t_2 de r_2 , deba haber una tupla t_1 de r_1 tal que $t_1[C_1] = t_2[\alpha]$. Las exigencias de este tipo se denominan **restricciones de integridad referencial**, o **dependencias de subconjuntos**.

```

create table cliente
  (nombre_cliente  char(20),
   calle_cliente   char(30),
   ciudad_cliente char(30),
   primary key (nombre_cliente))

create table sucursal
  (nombre_sucursal char(15),
   ciudad_sucursal char(30),
   activos          numeric(16,2),
   primary key (nombre_sucursal),
   check (activos >= 0))

create table cuenta
  (número_cuenta  char(10),
   nombre_sucursal char(15),
   saldo           numeric(12,2),
   primary key (número_cuenta),
   foreign key (nombre_sucursal) references sucursal,
   check (saldo >= 0))

create table impositor
  (nombre_cliente  char(20),
   número_cuenta  char(10),
   primary key (nombre_cliente, número_cuenta),
   foreign key (nombre_cliente) references cliente,
   foreign key (número_cuenta) references cuenta)

```

Figura 4.2 Definición de datos con SQL para parte de la base de datos bancaria.

El último término se debe a que la anterior restricción de integridad referencial puede escribirse como $\Pi_\alpha (r_2) \subseteq \Pi_{C_1} (r_1)$. Obsérvese que, para que las restricciones de integridad referencial tengan sentido, α y C_1 deben ser conjuntos de atributos compatibles, es decir, o bien α debe ser igual a C_1 , o bien deben contener el mismo número de atributos y los tipos de los atributos correspondientes deben ser compatibles (aquí se supone que α y C_1 están ordenados).

De manera predeterminada, en SQL las claves externas hacen referencia a los atributos de la clave primaria de la tabla referenciada. SQL también soporta una versión de la cláusula **references** en la que se puede especificar de manera explícita una lista de atributos de la relación a la que se hace referencia. La lista de atributos especificada, no obstante, debe declararse como clave candidata de la relación a la que hace referencia.

Se puede utilizar la siguiente forma abreviada como parte de la definición de atributos para declarar que el atributo forma una clave externa:

nombre_sucursal **char**(15) **references** sucursal

Cuando se viola una restricción de integridad referencial, el procedimiento normal es rechazar la acción que ha causado esa violación (es decir, la transacción que lleva a cabo la acción de actualización se retrocede). Sin embargo, la cláusula **foreign key** puede especificar que si una acción de borrado o de actualización de la relación a la que hace referencia viola la restricción, entonces, en lugar de rechazar la acción, el sistema realice los pasos necesarios para modificar la tupla de la relación que hace la referencia para que se restaure la restricción. Considérese esta definición de una restricción de integridad para la relación *cuenta*:

```
create table cuenta
(
    ...
    foreign key (nombre_sucursal) references sucursal
        on delete cascade
        on update cascade,
    ...
)
```

Debido a la cláusula **on delete cascade** asociada con la declaración de la clave externa, si el borrado de una tupla de *sucursal* da lugar a que se viole la restricción de integridad referencial, el sistema no rechaza el borrado. En vez de eso, el borrado “pasa en cascada” a la relación *cuenta*, borrando la tupla que hace referencia a la sucursal que se ha borrado. De manera parecida, el sistema no rechaza las actualizaciones de los campos a los que hace referencia la restricción aunque la violen; en vez de eso, el sistema actualiza también al nuevo valor el campo *nombre_sucursal* de las tuplas de *cuenta* que hacen la referencia. SQL también permite que la cláusula **foreign key** especifique acciones diferentes de **cascade**, si se viola la restricción. El campo que hace la referencia (en este caso, *nombre_sucursal*) puede definirse como nulo (empleando **set null** en lugar de **cascade**) o con el valor predeterminado para el dominio (utilizando **set default**).

Si hay una cadena de dependencias de clave externa que afecta a varias relaciones, el borrado o la actualización de un extremo de la cadena puede propagarse por toda ella. En el Ejercicio práctico 4.4 aparece un caso interesante en el que la restricción de **clave externa** de una relación hace referencia a la misma relación. Si una actualización o un borrado en cascada provoca una violación de la restricción que no pueda tratarse con otra operación en cascada, el sistema aborta la transacción. En consecuencia, todas las modificaciones provocadas por la transacción y sus acciones en cascada se deshacen.

Los valores **nulos** complican la semántica de las restricciones de integridad referencial en SQL. Se permite que los atributos de las claves externas sean valores nulos, siempre que no hayan sido declarados no nulos previamente. Si todas las columnas de una clave externa son no nulas en una tupla dada, se utiliza para esa tupla la definición habitual de las restricciones de clave externa. Si alguna de las columnas de la clave externa es nula, la tupla se define de manera automática para que satisfaga la restricción.

Puede que esta definición no sea siempre la opción correcta, por lo que SQL ofrece también constructores que permiten modificar el comportamiento cuando hay valores nulos; los constructores no se van a estudiar aquí.

Se pueden añadir restricciones de integridad a relaciones ya existentes utilizando el comando **alter table nombre-tabla add restricción**, donde *restricción* puede ser cualquiera de las restricciones que se han examinado. Cuando se ejecuta un comando de este tipo, el sistema se asegura primero de que la relación satisfaga la restricción especificada. Si lo hace, se añade la restricción a la relación; en caso contrario, se rechaza el comando.

Las transacciones pueden constar de varios pasos, y las restricciones de integridad pueden violarse temporalmente tras uno de los pasos, pero puede que uno posterior subsane la violación. Por ejemplo, supóngase la relación *persona* con la clave primaria *nombre*, y el atributo *cónyuge* y supóngase que *cónyuge* es una clave externa de *persona*. Es decir, la restricción impone que el atributo *cónyuge* debe contener un nombre que aparezca en la tabla *persona*. Supóngase que se desea destacar el hecho de que Martín y María están casados entre sí mediante la inserción de dos tuplas, una para cada uno, en la relación anterior. La inserción de la primera tupla violaría la restricción de clave externa, independientemente de cuál de las dos tuplas se inserte primero. Una vez insertada la segunda tupla, la restricción de clave externa vuelve a cumplirse.

Para tratar estas situaciones la norma de SQL permite que se añada la cláusula **initially deferred** a la especificación de la restricción; la restricción, en este caso, se comprueba al final de la transacción y no en los pasos intermedios¹. Las restricciones pueden especificarse de manera alternativa como **deferrable**, que significa que, de manera predeterminada, se comprueba inmediatamente, pero puede diferir si se desea. Para las restricciones declaradas como difieribles, la ejecución de la instrucción **set constraints**

1. Se puede evitar el problema del ejemplo anterior de otra manera, si el atributo *cónyuge* puede definirse como nulo: se definen los atributos *cónyuge* como nulos al insertar las tuplas de Martín y María y se actualizan posteriormente. Sin embargo, esta técnica es bastante confusa y no funciona si los atributos no pueden definirse como nulos.

lista-restricciones deferred como parte de una transacción hace que se difiera la comprobación de las restricciones especificadas hasta el final de esa transacción.

No obstante, hay que ser consciente de que el comportamiento predeterminado es comprobar las restricciones de manera inmediata, y de que muchas implementaciones no soportan la comprobación diferida de las restricciones.

4.2.6 Asertos

Un **aserto** es un predicado que expresa una condición que la base de datos debe satisfacer siempre. Las restricciones de dominio y las de integridad referencial son formas especiales de los asertos. Se ha prestado una atención especial a estos tipos de asertos porque se pueden verificar con facilidad y pueden afectar a una gran variedad de aplicaciones de bases de datos. Sin embargo, hay muchas restricciones que no se pueden expresar utilizando únicamente estas formas especiales. Ejemplos de estas restricciones son:

- La suma de los importes de todos los préstamos de cada sucursal debe ser menor que la suma de los saldos de todas las cuentas de esa sucursal.
- Cada préstamo tiene al menos un cliente que tiene una cuenta con un saldo mínimo de 1.000 €

En SQL los asertos adoptan la forma

```
create assertion <nombre-aserto> check <predicado>
```

En la Figura 4.3 se muestra cómo pueden escribirse estos dos ejemplos de restricciones en SQL. Dado que SQL no proporciona ningún mecanismo “para todo $X, P(X)$ ” (donde P es un predicado), no queda más remedio que implementarlo utilizando su equivalente “no existe X tal que no $P(X)$ ”, que puede escribirse en SQL.

Cuando se crea un aserto, el sistema comprueba su validez. Si el aserto es válido, sólo se permiten las modificaciones posteriores de la base de datos que no hagan que se viole el aserto. Esta comprobación puede introducir una sobrecarga importante si se han realizado asertos complejos. Por tanto, los asertos deben utilizarse con mucha cautela. La elevada sobrecarga debida a la comprobación y al mantenimiento de los asertos ha llevado a algunos desarrolladores de sistemas a soslayar el soporte para los asertos generales, o bien a proporcionar formas especializadas de aserto que resultan más sencillas de comprobar.

```
create assertion restricción_suma check
  (not exists (select * from sucursal
    where (select sum(importe) from préstamo
      where préstamo.nombre_sucursal = sucursal.nombre_sucursal)
    >= (select sum(saldo) from cuenta
      where cuenta.nombre_sucursal = sucursal.nombre_sucursal)))
```



```
create assertion restricción_saldo check
  (not exists (select * from préstamo
    where not exists (select *
      from prestatario, impositor, cuenta
      where préstamo.número_préstamo = prestatario.número_préstamo
        and prestatario.nombre_cliente = impositor.nombre_cliente
        and impositor.número_cuenta = cuenta.número_cuenta
        and cuenta.saldo >= 1000)))
```

Figura 4.3 Dos ejemplos de aserto.

4.3 Autorización

Se pueden asignar a los usuarios varios tipos de autorización para diferentes partes de la base de datos. Por ejemplo:

- La autorización de lectura
- La autorización de inserción
- La autorización de actualización
- La autorización de borrado

Cada uno de estos tipos de autorización se denomina **privilegio**. Se puede conceder a cada usuario todos estos tipos de privilegios, ninguno de ellos o una combinación de los mismos sobre partes concretas de la base de datos, como puede ser una relación o una vista.

La norma de SQL incluye los privilegios **select**, **insert**, **update** y **delete**. El privilegio **select** autoriza al usuario a leer los datos. Además de estas formas de privilegio para el acceso a los datos, SQL soporta otros privilegios, como el de crear, borrar o modificar relaciones y ejecutar procedimientos. Estos privilegios se estudiarán más adelante, en el Apartado 8.7. **all privileges** puede utilizarse como forma abreviada de todos los privilegios que se pueden conceder. El usuario que crea una relación nueva recibe de manera automática todos los privilegios sobre esa relación.

Se puede permitir al usuario al que se le ha concedido alguna forma de autorización que transmita (conceda) esa autorización a otros usuarios o que retire (revoque) una autorización concedida previamente.

El lenguaje de definición de datos de SQL incluye comandos para conceder y retirar privilegios. La instrucción **grant** se utiliza para conceder autorizaciones. La forma básica de esta instrucción es:

grant <lista de privilegios> **on** <nombre de relación o de vista> **to** <lista de usuarios o de roles>

La *lista de privilegios* permite la concesión de varios privilegios con un solo comando. El concepto de rol se trata más adelante, en el Apartado 8.7.

La siguiente instrucción **grant** concede a los usuarios de la base de datos Martín y María la autorización **select** sobre la relación *cuenta*:

grant select on cuenta to Martín, María

La autorización **update** puede concederse sobre todos los atributos de la relación o sólo sobre algunos. Si se incluye la autorización **update** en una instrucción **grant**, la lista de atributos sobre los que se concede la autorización **update** puede aparecer entre paréntesis justo después de la palabra clave **update**. Si se omite la lista de atributos, el privilegio **update** se concede sobre todos los atributos de la relación.

La siguiente instrucción **grant** concede a los usuarios Martín y María la autorización **update** sobre el atributo *importe* de la relación *préstamo*:

grant update (importe) on préstamo to Martín, María

El privilegio **insert** también puede especificar una lista de atributos; cualquier inserción en la relación debe especificar sólo esos atributos y el sistema asigna al resto de los atributos valores predeterminados (si hay alguno definido para ellos) o los define como nulos.

El nombre de usuario **public** hace referencia a todos los usuarios actuales y futuros del sistema. Por tanto, los privilegios concedidos a **public** se conceden de manera implícita a todos los usuarios actuales y futuros.

De manera predeterminada el usuario o rol al que se le concede un privilegio no está autorizado a concedérselo a otro usuario o rol. SQL permite que la concesión de privilegios especifique que el destinatario puede concedérselo, a su vez, a otro usuario. Esta característica se describe con más detalle en el Apartado 8.7.

Para retirar una autorización se emplea la instrucción **revoke**. Su forma es casi idéntica a la de **grant**:

```
revoke <lista de privilegios> on <nombre de la relación o nombre de la vista>
from <lista de usuarios o de roles>
```

Por tanto, para retirar los privilegios que se han concedido anteriormente:

```
revoke select on sucursal from Martín, María
revoke update (importe) on préstamo from Martín, María
```

La retirada de privilegios resulta más complicada si el usuario al que se le retiran los privilegios se los ha concedido a otros usuarios. Se volverá a este problema en el Apartado 8.7.

4.4 SQL incorporado

SQL proporciona un lenguaje de consultas declarativo muy potente. La formulación de consultas en SQL es normalmente mucho más sencilla que la formulación de las mismas en un lenguaje de programación de propósito general. Sin embargo, los programadores deben tener acceso a la base de datos desde los lenguajes de programación de propósito general, al menos, por dos razones:

1. No todas las consultas pueden expresarse en SQL, ya que SQL no ofrece todo el poder expresivo de los lenguajes de propósito general. Es decir, hay consultas que se pueden expresar en lenguajes como C, Java o Cobol que no se pueden expresar en SQL. Para formular consultas de este tipo, se puede incorporar SQL en un lenguaje más potente.
SQL está diseñado de tal forma que las consultas formuladas puedan optimizarse automáticamente y se ejecuten de manera eficiente—y proporcionar toda la potencia de los lenguajes de programación hace la optimización automática extremadamente difícil.
2. Las acciones no declarativas—como la impresión de informes, la interacción con los usuarios o el envío de los resultados de las consultas a una interfaz gráfica—no se pueden llevar a cabo desde el propio SQL. Normalmente, las aplicaciones contienen varios componentes y la consulta o actualización de los datos sólo es uno de ellos; los demás componentes se escriben en lenguajes de programación de propósito general. En el caso de las aplicaciones integradas, los programas escritos en el lenguaje de programación deben poder tener acceso a la base de datos.

La norma SQL define la incorporación de SQL en varios lenguajes de programación, tales como C, Cobol, Pascal, Java, PL/I, y Fortran. El lenguaje en el que se incorporan las consultas SQL se denomina lenguaje *anfitrión* y las estructuras de SQL que se admiten en el lenguaje anfitrión constituyen SQL *incorporado*.

Los programas escritos en el lenguaje anfitrión pueden usar la sintaxis de SQL incorporado para tener acceso a los datos almacenados en la base de datos y actualizarlos. Esta forma incorporada de SQL amplía aún más la capacidad de los programadores de manipular las bases de datos. En SQL incorporado toda la ejecución de las consultas la realiza el sistema de bases de datos, que luego pone el resultado de la consulta a disposición del programa tupla (registro) a tupla.

Los programas de SQL incorporado deben procesarlos un preprocesador especial antes de la compilación. El preprocesador sustituye las peticiones de SQL incorporado por declaraciones escritas en el lenguaje anfitrión y por llamadas a procedimientos que permiten la ejecución de los accesos a la base de datos en el momento de la ejecución. Posteriormente, el compilador del lenguaje anfitrión compila el programa resultante. Para identificar las peticiones al preprocesador de SQL incorporado se utiliza la instrucción EXEC SQL; tiene la forma

```
EXEC SQL <instrucción de SQL incorporado> END-EXEC
```

La sintaxis exacta de las peticiones de SQL incorporado depende del lenguaje en el que se haya incorporado SQL. Por ejemplo, cuando se incorpora SQL en C, se utiliza un punto y coma en lugar de END-EXEC. La incorporación de SQL en Java (denominada SQLJ) utiliza la sintaxis

```
# SQL { <instrucción de SQL incorporado> };
```

En el programa se incluye la instrucción SQL INCLUDE para identificar el lugar donde el preprocesador debe insertar las variables especiales que se emplean para la comunicación entre el programa y el sistema de bases de datos. Las variables del lenguaje anfitrión se pueden utilizar en las instrucciones de SQL incorporado, pero deben ir precedidas de dos puntos (:) para distinguirlas de las variables de SQL.

Antes de ejecutar ninguna instrucción de SQL el programa debe conectarse con la base de datos. Esto se logra mediante

```
EXEC SQL connect to servidor user nombre-usuario END-EXEC
```

En este caso, *servidor* identifica al servidor con el que hay que establecer la conexión. Algunas implementaciones de las bases de datos pueden exigir que se proporcione una contraseña además del nombre de usuario.

Las instrucciones de SQL incorporado son parecidas en cuanto a la forma a las instrucciones de SQL que se han descrito en este capítulo. Sin embargo, hay varias diferencias que se indican a continuación.

Para formular una consulta relacional se emplea la instrucción **declare cursor**. El resultado de la consulta no se calcula todavía. En vez de eso, el programa debe utilizar los comandos **open** y **fetch** (que se analizarán más adelante en este apartado) para obtener las tuplas resultado.

Considérese el esquema bancario que se ha utilizado en este capítulo. Supóngase la variable del lenguaje anfitrión *importe* y que se desea determinar el nombre y ciudad de residencia de los clientes que tienen más de *importe* euros en alguna de sus cuentas. Se puede escribir esta consulta del modo siguiente:

```
EXEC SQL
  declare c cursor for
    select nombre_cliente, ciudad_cliente
      from impositor,cliente,cuenta
     where impositor.nombre_cliente = cliente.nombre_cliente and
           cuenta.número_cliente = impositor.número_cuenta and
           cuenta.saldo > :importe
  END-EXEC
```

La variable *c* de la expresión anterior se denomina *cursor* de la consulta. Se utiliza esta variable para identificar la consulta en la instrucción **open**, que hace que se evalúe la consulta, y en la instrucción **fetch**, que hace que los valores de una tupla se coloquen en las variables del lenguaje anfitrión.

La instrucción **open** para la consulta de anterior es:

```
EXEC SQL open c END-EXEC
```

Esta instrucción hace que el sistema de base de datos ejecute la consulta y guarde el resultado en una relación temporal. La consulta tiene una variable del lenguaje anfitrión (:*importe*); la consulta utiliza el valor de la variable en el momento en que se ejecuta la instrucción **open**.

Si la consulta de SQL genera un error, el sistema de base de datos almacena un diagnóstico de error en las variables del área de comunicación de SQL (SQL communication-area, SQLCA), cuyas declaraciones inserta la instrucción SQL INCLUDE.

El programa de SQL incorporado ejecuta una serie de instrucciones **fetch** para recuperar las tuplas del resultado. La instrucción **fetch** necesita una variable del lenguaje anfitrión por cada atributo de la relación resultado. En la consulta de ejemplo se necesita una variable para almacenar el valor de *nombre_cliente* y otra para el de *ciudad_cliente*. Supóngase que esas variables son *nc* y *cc*, respectivamente. Entonces, la instrucción

```
EXEC SQL fetch c into :cn, :cc END-EXEC
```

genera una tupla de la relación resultado. El programa puede manipular entonces las variables *nc* y *cc* empleando las características del lenguaje de programación anfitrión.

Cada solicitud **fetch** devuelve una sola tupla. Para obtener todas las tuplas del resultado, el programa debe incorporar un bucle para iterar sobre todas las tuplas. SQL incorporado ayuda a los programadores en el tratamiento de esta iteración. Aunque las relaciones sean conceptualmente conjuntos, las tuplas del

resultado de las consultas se hallan en un orden físico determinado. Cuando el programa ejecuta una instrucción **open** sobre un cursor, ese cursor pasa a apuntar a la primera tupla del resultado. Cada vez que se ejecuta una instrucción **fetch**, el cursor se actualiza para que apunte a la siguiente tupla del resultado. Cuando no quedan por procesar más tuplas, la variable SQLSTATE de SQLCA se define como '02000' (que significa "sin datos"). Por tanto, se puede utilizar un bucle **while** (o su equivalente) para procesar cada tupla del resultado.

La instrucción **close** se debe utilizar para indicar al sistema de bases de datos que borre la relación temporal que guardaba el resultado de la consulta. Para el ejemplo anterior, esta instrucción tiene la forma

```
EXEC SQL close c END-EXEC
```

SQLJ, la incorporación en Java de SQL, ofrece una variación del esquema anterior, en la que se emplean los iteradores de Java en lugar de los cursosres. SQLJ asocia los resultados de la consulta con un iterador, y se puede utilizar el método **next()** del iterador de Java para pasar de una tupla a otra del resultado, igual que los ejemplos anteriores utilizan **fetch** sobre el cursor.

Las expresiones de SQL incorporado para la modificación (**update**, **insert** y **delete**) de las bases de datos no devuelven ningún resultado. Por tanto, son, de algún modo, más sencillas de expresar. Una solicitud de modificación de la base de datos tiene la forma

```
EXEC SQL < cualquier update, insert o delete válido > END-EXEC
```

En la expresión de SQL para la modificación de la base de datos pueden aparecer variables del lenguaje anfitrión precedidas de dos puntos. Si se produce un error en la ejecución de la instrucción, se establece un diagnóstico en SQLCA.

Las relaciones de las bases de datos también se pueden actualizar mediante cursosres. Por ejemplo, si se desea sumar 100 euros al atributo *saldo* de cada *cuenta* en la que el nombre de sucursal sea "Navacerrada", se puede declarar un cursor como sigue:

```
declare c cursor for
select *
from cuenta
where nombre_sucursal = 'Navacerrada'
for update
```

Después se itera por las tuplas ejecutando operaciones **fetch** sobre el cursor (como se mostró anteriormente) y, después de obtener cada tupla, se ejecuta el siguiente código:

```
update cuenta
set saldo = saldo + 100
where current of c
```

SQL incorporado permite que los programas en el lenguaje anfitrión tengan acceso a la base de datos, pero no proporciona ninguna ayuda para la presentación de los resultados al usuario ni para la generación de informes. La mayor parte de los productos comerciales de bases de datos incluyen herramientas para ayudar a los programadores de aplicaciones a crear interfaces de usuario e informes con formato. El Capítulo 8 describe la manera de crear aplicaciones de bases de datos con interfaces de usuario, concentrándose en las interfaces de usuario basadas en Web.

4.5 SQL dinámico

El componente *dinámico* de SQL permite que los programas construyan y remitan consultas de SQL en tiempo de ejecución. En cambio, las instrucciones de SQL incorporado deben hallarse presentes completamente en el momento de la compilación; las compila el preprocesador de SQL incorporado. Por medio de SQL dinámico los programas pueden crear consultas de SQL en tiempo de ejecución (quizás basadas en datos introducidos por el usuario) y hacer que se ejecuten inmediatamente o dejarlas *preparadas* para

su uso posterior. En el proceso de preparación de una instrucción SQL dinámica ésta se compila, y al usarla posteriormente se aprovecha su versión compilada.

SQL define normas para incorporar las llamadas dinámicas a SQL en el lenguaje anfitrión, por ejemplo, C, como se muestra en el siguiente ejemplo.

```
char * prog_SQL = "update cuenta set saldo = saldo *1.05
                     where número_cuenta = ?";
EXEC SQL prepare prog_din from :prog_SQL;
char cuenta[10] = "C-101";
EXEC SQL execute prog_din using :cuenta;
```

El programa SQL dinámico contiene un símbolo de interrogación ?, que representa un valor que se proporciona cuando se ejecuta el programa de SQL.

Sin embargo, esta sintaxis necesita extensiones para el lenguaje o un preprocesador para el lenguaje extendido. Una alternativa que se utiliza con frecuencia es emplear una API para enviar las consultas o actualizaciones de SQL a un sistema de bases de datos, sin realizar cambios en el propio lenguaje de programación.

En el resto de este apartado se examinan dos normas de conexión a bases de datos de SQL y la realización de consultas y de actualizaciones. Una, ODBC, es una interfaz para programas de aplicación desarrollada inicialmente para el lenguaje C, y extendida posteriormente a otros lenguajes como C++, C# y Visual Basic. La otra, JDBC, es una interfaz para programas de aplicación para el lenguaje Java.

Para entender estas normas hay que comprender el concepto de sesión de SQL. El usuario o la aplicación se *conectan* al servidor de SQL y establecen una sesión; ejecutan una serie de instrucciones y, finalmente, se *desconectan* de la sesión. Así, todas las actividades del usuario o de la aplicación están en el contexto de una sesión de SQL. Además de los comandos normales de SQL, las sesiones también pueden incluir comandos para *comrometer* el trabajo realizado en la sesión o para *retrocederlo*.

4.5.1 ODBC

La norma **ODBC** (Open Database Connectivity, conectividad abierta de bases de datos) define el modo de comunicación entre los programas de aplicación y los servidores de bases de datos. ODBC define una **interfaz para programas de aplicación** (API, Application Program Interface) que pueden utilizar las aplicaciones para abrir conexiones con las bases de datos, enviar las consultas y las actualizaciones y obtener los resultados. Las aplicaciones como las interfaces gráficas de usuario, los paquetes estadísticos y las hojas de cálculo pueden emplear la misma API de ODBC para conectarse a cualquier servidor de bases de datos compatible con ODBC.

Cada sistema de bases de datos compatible con ODBC proporciona una biblioteca que se debe enlazar con el programa cliente. Cuando el programa cliente realiza una llamada a la API de ODBC, el código de la biblioteca se comunica con el servidor para llevar a cabo la acción solicitada y obtener los resultados.

La Figura 4.4 muestra un ejemplo de código C que usa la API de ODBC. El primer paso en el uso de ODBC para comunicarse con un servidor es configurar su conexión. Para ello, el programa asigna en primer lugar un entorno de SQL, luego un manejador para la conexión a la base de datos. ODBC define los tipos HENV, HDBC y RETCODE. El programa abre a continuación la conexión a la base de datos empleando SQLConnect. Esta llamada tiene varios parámetros, como son el manejador de la conexión, el servidor al que hay que conectarse, el identificador de usuario y la contraseña para la base de datos. La constante SQL_NTS indica que el argumento anterior es una cadena terminada en un valor nulo.

Una vez configurada la conexión, el programa puede enviar comandos SQL a la base de datos empleando SQLExecDirect. Las variables del lenguaje C se pueden vincular a los atributos del resultado de la consulta, de forma que cuando se obtenga una tupla resultado mediante SQLFetch, los valores de sus atributos se almacenen en las variables de C correspondientes. La función SQLBindCol realiza esta tarea; el segundo argumento identifica la posición del atributo en el resultado de la consulta, y el tercer argumento indica la conversión de tipos de SQL a C requerida. El siguiente argumento proporciona la dirección de la variable. Para los tipos de datos de longitud variable como los arrays de caracteres, los dos últimos argumentos proporcionan la longitud máxima de la variable y una ubicación donde se debe almacenar la longitud real cuando se obtenga una tupla. Un valor negativo devuelto para el argumento

```

void ODBCexample()
{
    RETCODE error;
    HENV ent; /* entorno */
    HDBC con; /* conexión a la base de datos */

    SQLAllocEnv(&ent);
    SQLAllocConnect(ent, &con);
    SQLConnect(con, "db.yale.edu", SQL_NTS, "avi", SQL_NTS,
               "contraseñaAvi", SQL_NTS);
{
    char nombresucursal[80];
    float saldo;
    int lenOut1, lenOut2;
    HSTMT stmt;

    char * consulta = "select nombre_sucursal, sum (saldo)
                       from cuenta
                       group by nombre_sucursal";
    SQLAllocStmt(con, &stmt);
    error = SQLExecDirect(stmt, consulta, SQL_NTS);
    if (error == SQL_SUCCESS) {
        SQLBindCol(stmt, 1, SQL_C_CHAR, nombresucursal , 80, &lenOut1);
        SQLBindCol(stmt, 2, SQL_C_FLOAT, &saldo, 0 , &lenOut2);
        while (SQLFetch(stmt) == SQL_SUCCESS) {
            printf ("%s %g\n", nombresucursal, saldo);
        }
    }
    SQLFreeStmt(stmt, SQL_DROP);
}
SQLDisconnect(con);
SQLFreeConnect(con);
SQLFreeEnv(ent);
}

```

Figura 4.4 Código de ejemplo ODBC.

de la longitud indica que el valor es **null**. Para los tipos de datos de longitud fija como integer o float, se ignora el argumento que indica la longitud máxima, mientras que la devolución de un valor negativo para el argumento de la longitud indica un valor nulo.

La instrucción `SQLFetch` se halla en un bucle `while` que se ejecuta hasta que `SQLFetch` devuelva un valor diferente de `SQL_SUCCESS`. En cada iteración, el programa almacena los valores en variables de C como se especifica mediante las llamadas a `SQLBindCol` e imprime esos valores.

Al final de la sesión, el programa libera el controlador de la instrucción, se desconecta de la base de datos y libera la conexión y los manejadores del entorno de SQL. Un buen estilo de programación exige que el resultado de cada llamada a una función se compruebe para asegurarse de que no haya errores; se han omitido la mayoría de estas comprobaciones en aras de la brevedad.

Es posible crear instrucciones SQL con parámetros; por ejemplo, considérese la instrucción `insert into cuenta values(?, ?, ?)`. Los signos de interrogación representan los valores que se proporcionarán después. Esta instrucción se puede “preparar”, es decir, compilar en la base de datos y ejecutar repetidamente proporcionando los valores reales para los parámetros—en este caso, proporcionando un número de cuenta, un nombre de sucursal y un saldo para la relación *cuenta*.

ODBC define funciones para gran variedad de tareas, tales como hallar todas las relaciones de la base de datos y los nombres y tipos de las columnas del resultado de una consulta o de una relación de la base de datos.

De forma predeterminada, cada instrucción SQL se trata como una transacción separada que se compromete automáticamente. La llamada `SQLSetConnectOption(con, SQL_AUTOCOMMIT, 0)` desactiva el compromiso automático en la conexión `con`, por lo que las transacciones se deben comprometer explícitamente mediante `SQLTransact(con, SQL_COMMIT)` o retroceder mediante `SQLTransact(con, SQL_ROLLBACK)`.

La norma ODBC define *niveles de conformidad*, que especifican subconjuntos de la funcionalidad definida por la norma. Puede que una implementación de ODBC sólo proporcione las características básicas, o puede proporcionar características más avanzadas (de nivel 1 o de nivel 2). El nivel 1 exige soporte para la captura de información sobre el catálogo, como puede ser la información sobre las relaciones existentes y los tipos de sus atributos. El nivel 2 exige más características, como la capacidad de enviar y obtener arrays de valores de parámetros y la de obtener información del catálogo más detallada.

La norma de SQL define una **interfaz del nivel de llamadas** (CLI, Call Level Interface) que es parecida a la interfaz de ODBC. Las APIs ADO y ADO.NET son alternativas a ODBC, diseñadas para los lenguajes Visual Basic y C#; consultense las notas bibliográficas si se desea más información.

La norma de JDBC define una API que pueden utilizar los programas de Java para conectarse a los servidores de bases de datos (la palabra JDBC era originalmente una abreviatura de **Java Database Connectivity**—conectividad de bases de datos con Java—, pero la forma completa ya no se emplea).

4.5.1.1 Apertura de conexiones y ejecución de consultas

La Figura 4.5 muestra un ejemplo de programa Java que utiliza la interfaz JDBC. El programa, en primer lugar, debe abrir una conexión con la base de datos y después puede ejecutar las instrucciones de SQL, pero antes de abrir la conexión, carga los controladores correspondientes para la base de datos empleando `Class.forName`. El primer parámetro de la llamada `getConnection` especifica el nombre de la máquina en la que se ejecuta el servidor (en este caso, `db.yale.edu`) y el número del puerto que emplea para las comunicaciones (en este caso, `2.000`). El parámetro también especifica el esquema de la base de datos que se va a utilizar (en este caso, `bdbanco`), ya que cada servidor de bases de datos puede dar soporte a varios esquemas. El primer parámetro también especifica el protocolo que se va a utilizar para la comunicación con la base de datos (en este caso, `jdbc:oracle:thin:`). Obsérvese que JDBC especifica sólo la API, no el protocolo de comunicación. Los controladores de JDBC pueden dar soporte a varios protocolos y se debe especificar alguno compatible tanto con la base de datos como con el controlador. Los otros dos argumentos de `getConnection` son un identificador de usuario y una contraseña.

4.5.2 JDBC

El programa crea a continuación un manejador de instrucciones para la conexión y lo usa para ejecutar una instrucción SQL y obtener los resultados. En nuestro ejemplo, `stmt.executeUpdate` ejecuta una instrucción de actualización. El constructor `try { ... } catch { ... }` permite capturar cualquier excepción (condición de error) que surja cuando se realicen las llamadas JDBC, e imprime el mensaje correspondiente para el usuario.

El programa puede ejecutar una consulta mediante `stmt.executeQuery`. Puede recuperar el conjunto de filas del resultado en `ResultSet` y capturarlas tupla a tupla empleando la función `next()` en el conjunto de resultados. La Figura 4.5 muestra dos formas de recuperar los valores de los atributos de las tuplas: empleando el nombre del atributo (`nombre_sucursal`) y utilizando la posición del atributo (2, para denotar el segundo atributo).

La conexión se cierra al final del procedimiento. Obsérvese que es importante cerrar la conexión, ya que se ha impuesto un límite al número de conexiones con la base de datos; las conexiones no cerradas pueden hacer que ese límite se supere. Si esto ocurre, la aplicación no podrá abrir más conexiones con la base de datos.

```

public static void JDBCexample(String dbid, String idusuario, String contraseña)
{
    try {
        Class.forName ("oracle.jdbc.driver.OracleDriver");
        Connection con = DriverManager.getConnection(
            "jdbc:oracle:thin:@db.yale.edu:2000:bdbanco",
            idusuario, contraseña);
        Statement stmt = con.createStatement();
        try {
            stmt.executeUpdate(
                "insert into cuenta values('C-9732', 'Navacerrada', 1200)");
        } catch (SQLException sqle)
        {
            System.out.println("No se pudo insertar la tupla. "+ sqle);
        }
        ResultSet rset = stmt.executeQuery(
            "select nombre_sucursal, avg (saldo)
             from cuenta
             group by nombre_sucursal");
        while (rset.next()) {
            System.out.println(rset.getString("nombre_sucursal") + +
                               rset.getFloat(2));
        }
        stmt.close();
        con.close();
    }
    catch (SQLException sqle)
    {
        System.out.println("Excepción de SQL : "+ sqle);
    }
}

```

Figura 4.5 Un ejemplo de código JDBC.

4.5.2.1 Instrucciones preparadas

Se pueden crear instrucciones preparadas en las que algunos valores estén sustituidos por “?”, lo que indica que los valores reales se proporcionarán más tarde. Algunos sistemas de bases de datos compilan las consultas cuando se preparan; cada vez que se ejecuta la consulta (con valores nuevos), la base de datos puede volver a emplear la forma previamente compilada de la consulta. El fragmento de código de la Figura 4.6 muestra la manera de utilizar las instrucciones preparadas. La función `setString` (y otras funciones parecidas para otros tipos de datos básicos de SQL) permite especificar el valor de los parámetros.

Las instrucciones preparadas son el método preferido de ejecución de las consultas de SQL, cuando utilizan valores que introducen los usuarios. Supóngase que el valor de las variables `número_cuenta`, `nombre_sucursal`, and `saldo` ha sido introducido por un usuario y hay que insertar la fila correspondiente en la relación `cuenta`. Supóngase que, en lugar de utilizar una instrucción preparada, se crea una consulta mediante la concatenación de las cadenas de caracteres de la manera siguiente:

```
"insert into cuenta values(' "+ número_cuenta + ', ' "+ nombre_sucursal + ',
                           + saldo + ")"
```

y la consulta se ejecuta directamente. Ahora, si el usuario escribiera una sola comilla en el campo del número de cuenta o en el del nombre de la sucursal, la cadena de caracteres de la consulta tendría un error

```

PreparedStatement pStmt = con.prepareStatement(
    "insert into cuenta values(?, ?, ?)");
pStmt.setString(1, "C-9732");
pStmt.setString(2, "Navacerrada");
pStmt.setInt(3, 1200);
pStmt.executeUpdate();
pStmt.setString(1, "C-9733");
pStmt.executeUpdate();

```

Figura 4.6 Instrucciones preparadas en código JDBC.

de sintaxis. Es bastante probable que alguna sucursal tenga un apóstrofo en su nombre (especialmente si es un nombre irlandés como O'Donnell). Peor todavía, intrusos informáticos maliciosos pueden “inyectar” consultas de SQL propias escribiendo los caracteres adecuados en la cadena de caracteres. Una inyección de SQL de ese tipo puede dar lugar a graves fallos de seguridad.

La adición de caracteres de escape para tratar con los apóstrofos de las cadenas de caracteres es una manera de resolver este problema. El uso de las instrucciones preparadas es una manera más sencilla de hacerlo, ya que el método `setString` añade caracteres de escape de manera implícita. Además, cuando hay que ejecutar varias veces la misma instrucción con valores diferentes, las instrucciones preparadas suelen ejecutarse mucho más rápido que varias instrucciones de SQL por separado.

JDBC también proporciona una interfaz `CallableStatement` que permite la llamada a los procedimientos almacenados y a las funciones de SQL (que se describen más adelante, en el Apartado 4.6). Esta interfaz desempeña el mismo rol para las funciones y los procedimientos que `prepareStatement` para las consultas.

```

CallableStatement cStmt1 = con.prepareCall("{? = call alguna_función(?)}");
CallableStatement cStmt2 = con.prepareCall("{call algún_procedimiento(?)}");

```

Los tipos de datos de los valores devueltos por las funciones y de los parámetros externos de los procedimientos deben registrarse empleando el método `registerOutParameter()`, y pueden recuperarse utilizando métodos `get` parecidos a los de los conjuntos de resultados.

4.5.2.2 Metadatos

JDBC también proporciona mecanismos para examinar los esquemas de las bases de datos y determinar el tipo de datos de los atributos de los conjuntos de resultados. La interfaz `ResultSet` tiene un método `getMetaData()` para obtener objetos `ResultSetMetaData` y proporcionar los metadatos del conjunto de resultados. La interfaz `ResultSetMetaData`, a su vez, contiene métodos para determinar la información de los metadatos, como puede ser el número de columnas de un resultado, el nombre de una columna concreta o el tipo de datos de una columna dada. El programa de JDBC que se muestra a continuación ilustra el uso de la interfaz `ResultSetMetaData` para mostrar el nombre y el tipo de datos de todas las columnas de un conjunto de resultados. Se da por supuesto que la variable `cr` del siguiente código es un conjunto de resultados obtenido mediante la ejecución de una consulta.

```

ResultSetMetaData rsmd = cr.getMetaData();
for(int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i));
    System.out.println(rsmd.getColumnTypeName(i));
}

```

El interfaz `DatabaseMetaData` ofrece una manera de determinar los metadatos de la base de datos. La interfaz `Connection` contiene un método `getMetaData` que devuelve el objeto `DatabaseMetaData`. La interfaz `DatabaseMetaData`, a su vez, contiene gran número de métodos para obtener los metadatos de la base de datos. El código de la Figura 4.7 ilustra la manera de obtener información sobre las columnas (atributos) de las relaciones de las bases de datos. Se da por supuesto que la variable `con` almacena una

```

DatabaseMetaData dbmd = con.getMetaData();
ResultSet sr = dbmd.getColumns(null, "bdbanco", "cuenta", "%");
    // Argumentos para getColumns: Catálogo, Patrón-esquema, Patrón-tabla y
    //           Patrón-columna
    // Devuelve: Una fila por columna; cada fila tiene varios atributos, como
    //           COLUMN_NAME (nombre de la columna), TYPE_NAME (nombre del tipo)
while( sr.next()) {
    System.out.println(sr.getString("COLUMN_NAME"), sr.getString("TYPE_NAME"));
}

```

Figura 4.7 Búsqueda de información sobre las columnas en JDBC empleando DatabaseMetaData.

conexión con la base de datos ya abierta. El método `getColumns` toma cuatro argumentos: un nombre de catálogo (null significa que hay que ignorar el nombre del catálogo), un patrón para el nombre de los esquemas, un patrón para el nombre de las tablas y un patrón para el nombre de las columnas. El nombre del esquema, el de la tabla y los patrones del nombre de las columnas pueden utilizarse para especificar un nombre o un patrón. Los patrones pueden emplear los caracteres especiales para encaje de cadenas de caracteres de SQL "%" y "_"; por ejemplo, el patrón "%" encaja con todos los nombres.

Sólo se recuperan las columnas de las tablas de los esquemas que satisfacen el nombre o el patrón especificados. Cada fila del conjunto de resultados contiene información sobre una columna. Las filas tienen varias columnas, como pueden ser el nombre del catálogo, del esquema, de la tabla y de la columna, el tipo de datos de la columna, etc.

Otros métodos proporcionados por `DatabaseMetaData` permiten la recuperación de metadatos de las relaciones (`getTables()`), las referencias a claves externas (`getCrossReference()`), las autorizaciones, los límites de las bases de datos como el número máximo de conexiones, etc.

Las interfaces para metadatos pueden emplearse para gran variedad de tareas. Por ejemplo, se pueden utilizar par escribir un navegador de la base de datos que permita a los usuarios buscar las tablas de las bases de datos, examinar su esquema, examinar las filas de las tablas, aplicar selecciones para ver las filas deseadas, etc. La información sobre los metadatos puede utilizarse para crear código empleado para estas tareas genéricas; por ejemplo, se puede escribir código para mostrar las filas de las relaciones de manera que funcione en todas las relaciones posibles, independientemente de su esquema. De manera parecida, es posible escribir código que tome una cadena de caracteres de consulta, ejecute la consulta e imprima el resultado en forma de tabla con formato; el código puede funcionar independientemente de la consulta concreta que se remita.

4.5.2.3 Otras características

JDBC ofrece varias características más, como los **conjuntos de resultados actualizables**. Se pueden crear conjuntos de resultados actualizables a partir de consultas que lleven a cabo selecciones o proyecciones de las relaciones de las bases de datos. La actualización de una tupla del conjunto de resultados tiene como consecuencia que se actualice la tupla correspondiente de la relación de la base de datos.

De manera predeterminada, cada instrucción de SQL se trata como una transacción independiente que se compromete de manera automática. El método `setAutoCommit()` de la interfaz `Connection` de JDBC permite que este comportamiento se active o se desactive. Por tanto, si `con` es una variable que almacena una conexión abierta, `conn.setAutoCommit(false)` desactivará el compromiso automático. Las transacciones, entonces, deben comprometerse de manera explícita mediante `con.commit()` o retrocederse mediante `con.rollback()`. El compromiso automático puede activarse mediante `con.setAutoCommit(true)`.

JDBC proporciona interfaces para el tratamiento de objetos de gran tamaño sin necesidad de crearlos completamente en la memoria. Para capturar objetos de gran tamaño la clase `ResultSet` proporciona los métodos `getBlob()` y `getBlob()`, que son parecidos al método `getString()` pero devuelven objetos de los tipos de datos `Blob` y `Clob`, respectivamente. Estos objetos no almacenan los objetos de gran tamaño completos, sino que almacenan sus localizadores. Las clases `Blob` y `Clob` proporcionan métodos para recuperar objetos de gran tamaño en fragmentos más pequeños. También permiten que se almacenen objetos de gran tamaño en la base de datos; pueden asociarse con corrientes de datos de Java, que se

```

create function recuento_cuentas(nombre_cliente varchar(20))
    returns integer
    begin
        declare recuento_c integer;
        select count(*) into recuento_c
        from impositor
        where impositor.nombre_cliente = nombre_cliente
        return recuento_c;
    end

```

Figura 4.8 Función definida en SQL.

capturan de manera transparente y se envían a la base de datos en fragmentos de pequeño tamaño, por lo que no hace falta crear el objeto completo en la memoria.

JDBC también proporciona la clase RowSet, que ofrece las mismas características de ResultSet y varias características adicionales. Para obtener más información sobre JDBC, consultese la información bibliográfica del final del capítulo.

4.6 Funciones y procedimientos**

Desde la versión SQL:1999, SQL permite la definición de funciones, procedimientos y métodos. Pueden definirse bien mediante el componente procedimental de SQL:1999 o bien mediante un lenguaje de programación externo como Java, C o C++. En primer lugar se examinarán las definiciones en SQL y luego se verá la manera de utilizar las definiciones en lenguajes externos.

Varios sistemas de bases de datos soportan sus propias extensiones procedimentales de SQL, como PL/SQL en Oracle y TransactSQL en Microsoft SQL Server. Estas extensiones recuerdan la parte procedimental de SQL, pero puede haber diferencias significativas en sintaxis y en semántica; consultense los manuales de los sistemas respectivos para obtener más detalles.

4.6.1 Funciones y procedimientos de SQL

Supóngase que se desea una función que, dado el nombre de un cliente, devuelva el número de cuentas de las que es titular ese cliente. Se puede definir la función como se muestra en la Figura 4.8.

Esta función puede utilizarse en una consulta que devuelva el nombre y la dirección de todos los clientes con más de una cuenta:

```

select nombre_cliente, calle_cliente, ciudad_cliente
from cliente
where recuento_cuentas(nombre_cliente) > 1

```

Las funciones resultan particularmente útiles con tipos de datos especializados tales como las imágenes y los objetos geométricos. Por ejemplo, un tipo de datos de segmentos de recta empleado en las bases de datos cartográficas puede tener una función asociada que compruebe si se solapan dos segmentos, y un tipo de datos de imagen puede tener asociadas funciones para comparar la semejanza de dos imágenes. Las funciones se pueden escribir en un lenguaje externo como C, como se vio en el Apartado 4.6.3.

Desde la versión SQL:2003, SQL soporta las funciones que pueden devolver tablas como resultado; esas funciones se denominan **funciones de tabla**. Considérese la función definida en la Figura 4.9. Esta función devuelve una tabla que contiene todas las cuentas que tiene una persona dada. Obsérvese que se hace referencia a los parámetros de las funciones anteponiéndoles el nombre de la función (*cuentas_de.nombre_cliente*).

La función puede emplearse en consultas de la manera siguiente:

```

select *
from table(cuentas_de('Gómez'))

```

```

create function cuentas_de (nombre_cliente char(20))
    returns table (
        número_cuenta char(10),
        nombre_sucursal char(15),
        saldo numeric(12,2))
return table
    (select número_cuenta, nombre_sucursal, saldo
     from cuenta
     where exists (
         select *
         from impositor
         where impositor.nombre_cliente = cuentas_de.nombre_cliente and
               impositor.número_cuenta = cuenta.número_cuenta)
    )
)

```

Figura 4.9 Función de tabla en SQL.

Esta consulta devuelve todas las cuentas pertenecientes al cliente 'Gómez'. En el sencillo caso anterior, resulta fácil escribir esta consulta sin emplear funciones evaluadas sobre tablas. En general, sin embargo, las funciones evaluadas sobre tablas pueden considerarse **vistas paramétricas** que generalizan el concepto habitual de las vistas permitiendo la introducción de parámetros.

SQL:1999 también permite el uso de procedimientos. La función *recuento_cuentas* se puede escribir en forma de procedimiento:

```

create procedure proc_recuento_cuentas(in nombre_cliente varchar(20),
                                       out recuento_c integer)
begin
    select count(*) into recuento_
    from impositor
    where impositor.nombre_cliente = proc_recuento_cuentas.nombre_cliente
end

```

Se pueden invocar los procedimientos mediante la instrucción **call** desde otro procedimiento de SQL o desde SQL incorporado:

```

declare recuento_cuentas integer;
call proc_recuento_cuentas('Gómez', recuento_c);

```

Los procedimientos y las funciones pueden invocarse desde SQL dinámico, como se ilustra mediante la sintaxis de JDBC en el Apartado 4.5.2.2.

SQL:1999 permite que haya más de un procedimiento con el mismo nombre, siempre que el número de argumentos de los procedimientos con el mismo nombre sea diferente. El nombre, junto con el número de argumentos, se utiliza para identificar los procedimientos. SQL permite también más de una función con el mismo nombre, siempre que las diferentes funciones con el mismo nombre tengan diferente número de argumentos o, para funciones con el mismo número de argumentos, se diferencien en el tipo de datos de un argumento, como mínimo.

4.6.2 Constructores procedimentales

Desde la versión SQL:1999, SQL soporta varios constructores procedimentales que le proporcionan casi toda la potencia de los lenguajes de programación de propósito general. La parte de la norma SQL que trata de estos constructores se denomina **módulo de almacenamiento persistente** (Persistent Storage Module, PSM).

El objetivo de PSM en SQL no es sustituir a los lenguajes de programación convencionales. Más bien, los constructores procedimentales permiten que se registre la "lógica de negocio" en forma de procedimientos almacenados de la base de datos y se ejecute en propia la base de datos. Por ejemplo, los bancos

suelen tener muchas reglas sobre la manera y el momento en que se pueden hacer los pagos a los clientes, como los límites máximos de retirada de efectivo, las exigencias de saldo mínimo, las facilidades para los descubiertos que permiten a los clientes retirar más dinero que el saldo disponible mediante la concesión automática de un préstamo, etc.

Aunque esa lógica de negocio puede codificarse en forma de procedimientos de los lenguajes de programación almacenados completamente fuera de las bases de datos, su definición como procedimientos almacenados de la base de datos presenta varias ventajas. Por ejemplo, permite que varias aplicaciones tengan acceso a los procedimientos, así como que baste con modificar un solo punto en caso de cambio de las reglas de negocio, sin que haga falta modificar la aplicación. El código de la aplicación puede llamar a los procedimientos almacenados, en lugar de actualizar directamente las relaciones de la base de datos.

Los constructores procedimentales son necesarios para permitir que se codifiquen como procedimientos almacenados las reglas de negocio complejas y, por tanto, se añadieron a SQL a partir de la versión SQL:1999 (ya las soportaban algunos productos de bases de datos incluso antes).

Las instrucciones compuestas son de la forma **begin** ... **end**, y pueden contener varias instrucciones de SQL entre **begin** y **end**. En las instrucciones compuestas pueden declararse variables locales, como se ha visto en el Apartado 4.6.1.

SQL:1999 soporta las instrucciones **while** y **repeat** con la sintaxis siguiente:

```
declare n integer default 0;
while n < 10 do
    set n = n + 1;
end while
repeat
    set n = n - 1;
until n = 0
end repeat
```

Este código no lleva a cabo nada útil; simplemente pretende mostrar la sintaxis de los bucles **while** y **repeat**. Posteriormente se verán aplicaciones más significativas.

También hay un bucle **for** que permite la iteración por todos los resultados de una consulta:

```
declare n integer default 0;
for r as
    select saldo from cuenta
    where nombre_sucursal = 'Navacerrada'
do
    set n = n + r.saldo
end for
```

El programa abre de manera implícita un cursor cuando comienza la ejecución del bucle **for** y lo utiliza para capturar los valores fila a fila en la variable de bucle **for** (*r*, en este ejemplo). Es posible ponerle nombre al cursor, insertando el texto *nc cursor for* justo detrás de la palabra clave **as**, donde *nc* es el nombre que se le desea dar al cursor. El nombre del cursor puede emplearse para llevar a cabo operaciones de actualización o borrado sobre la tupla a la que apunte el cursor. La instrucción **leave** se puede emplear para salir del bucle, mientras que **iterate** comienza con la siguiente tupla, a partir del comienzo del bucle, y se salta las instrucciones restantes.

Entre las instrucciones condicionales soportadas por SQL están las instrucciones if-then-else con la sintaxis:

```

if r.saldo < 1000
    then set l = l + r.saldo
elseif r.saldo < 5000
    then set m = m + r.saldo
else set h = h + r.saldo
end if

```

Este código da por supuesto que *l*, *m* y *h* son variables enteras y que *r* es una variable de fila. Si se sustituye la línea “**set n = n + r.saldo**” del bucle **for** del párrafo anterior por el código **if-then-else** (y se proporcionan las declaraciones y los valores iniciales correspondientes de *l*, *m* y *h*), el bucle calculará los saldos totales de las cuentas que se hallan en las categorías de saldos bajo, medio y alto, respectivamente.

SQL también soporta instrucciones **case** parecidas a la instrucción **case** del lenguaje C/C++ (además de las expresiones **case**, que se vieron en el Capítulo 3).

Finalmente, SQL incluye los conceptos de señalización de las **condiciones de excepción** y de declarar los **manejadores** que pueden tratar esa excepción, como en este código:

```

declare agotado condition
declare exit handler for agotado
begin
    ...
end

```

Las instrucciones entre **begin** y **end** pueden provocar una excepción ejecutando **signal agotado**. El manejador dice que si se da la condición, la acción que hay que tomar es salir de la instrucción **begin end** circundante. Una acción alternativa sería **continue**, que continúa con la ejecución a partir de la siguiente instrucción que ha generado la excepción. Además de las condiciones definidas de manera explícita, también hay condiciones predefinidas como **sqlexception**, **sqlwarning** y **not found**.

La Figura 4.10 proporciona un ejemplo de mayor tamaño del uso de constructores procedimentales en SQL. El procedimiento *retirar* definido en la figura retira dinero de una cuenta y, si el saldo pasa a ser negativo, comienza el tratamiento del descubierto; no se muestra el código para el tratamiento del descubierto. La función devuelve un código de error; con un valor mayor o igual que cero indica éxito, mientras que un valor negativo indica una condición de error.

Otro ejemplo que ilustra los bucles **while** se presenta más adelante, en el Apartado 4.7.

4.6.3 Rutinas en otros lenguajes

SQL permite definir funciones en lenguajes de programación como Java, C#, C o C++. Las funciones definidas de esta manera pueden ser más eficientes que las definidas en SQL, y cálculos que no pueden llevarse a cabo en SQL pueden ejecutarse mediante estas funciones. Un ejemplo del uso de este tipo de funciones es llevar a cabo un cálculo aritmético complejo sobre los datos de una tupla.

Los procedimientos y funciones externos pueden especificarse de esta manera:

```

create procedure proc_recuento_cuentas( in nombre_cliente varchar(20),
                                         out count integer)
language C
external name '/usr/avi/bin/proc_recuento_cuentas'

create function recuento_cuentas (nombre_cliente varchar(20))
returns integer
language C
external name '/usr/avi/bin/recuento_cuentas'

```

Los procedimientos del lenguaje externo tienen que tratar con los valores nulos y con las excepciones. Por tanto, deben tener varios parámetros adicionales: un valor **sqlstate** para indicar el estado de éxito o de fracaso, un parámetro para almacenar el valor devuelto por la función y variables indicadoras para el resultado de cada parámetro o función, para indicar si su valor es nulo. Una línea adicional **parameter**

```

create procedure retirar(
    in número_cuenta varchar(10)
    in importe numeric(12,2))
-- retira dinero de una cuenta
returns integer
begin
    declare nuevosaldo numeric(12,2);
    select saldo into nuevosaldo
    from cuenta
    where cuenta.número_cuenta = retirar.número_cuenta;
    nuevosaldo = nuevosaldo - importe;
    if (nuevosaldo < 0)
        begin
            ... el código para manejar el descubierto se inserta aquí
            ... si el importe es demasiado elevado para que lo maneje descubierto
            ... devuelve el código de error -1
        end
    else begin
        update cuenta
            set saldo = saldo - nuevosaldo
            where cuenta.número_cuenta = retirar.número_cuenta
    end
    return(0);
end

```

Figura 4.10 Procedimiento para la retirada de dinero de las cuentas.

style general añadida a la declaración anterior indica que los procedimientos o funciones externos sólo toman los argumentos mostrados y no tratan con valores nulos ni excepciones.

Las funciones definidas en un lenguaje de programación y compiladas fuera del sistema de base de datos pueden cargarse y ejecutarse con el código del sistema de bases de datos. Sin embargo, ello conlleva el riesgo de que un fallo del programa corrompa las estructuras internas de la base de datos y pueda saltarse la funcionalidad de control de accesos del sistema de bases de datos. Los sistemas de bases de datos que se preocupan más del rendimiento eficiente que de la seguridad pueden ejecutar los procedimientos de este modo. Los sistemas de bases de datos que están preocupados por la seguridad pueden ejecutar ese código como parte de un proceso diferente, comunicarle el valor de los parámetros y recuperar los resultados mediante la comunicación entre procesos. Sin embargo, el tiempo añadido que supone la comunicación entre los procesos es bastante elevado, en las arquitecturas de CPU habituales, decenas a centenares de millares de instrucciones se pueden ejecutar en el tiempo necesario para una comunicación entre los procesos.

Si el código está escrito en un lenguaje “seguro” como Java o C#, cabe otra posibilidad: ejecutar el código en un **cubo** dentro del propio proceso de ejecución de la consulta a la base de datos. El cubo permite que el código de Java o de C# tenga acceso a su propia área de memoria, pero evita que ese código lea o actualice la memoria del proceso de ejecución de la consulta, o tenga acceso a los archivos del sistema de archivos (la creación de cubos no es posible en lenguajes como C, que permite el acceso sin restricciones a la memoria mediante los punteros). El evitar la comunicación entre procesos reduce enormemente la sobrecarga de llamadas a funciones.

Varios sistemas de bases de datos actuales soportan que las rutinas de lenguajes externos se ejecuten en cubos dentro del proceso de ejecución de las consultas. Por ejemplo, Oracle y DB2 de IBM permiten que las funciones de Java se ejecuten como parte del proceso de la base de datos. SQL Server 2005 de Microsoft permite que los procedimientos compilados en CLR (Common Language Runtime) se ejecuten dentro del proceso de la base de datos; esos procedimientos pueden haberse escrito, por ejemplo, en C# o en Visual Basic.

nombre_empleado	nombre_jefe
Alández	Bariego
Bariego	Erice
Corisco	Dalma
Dalma	Santos
Erice	Santos
Santos	Marchamalo
Rienda	Marchamalo

Figura 4.11 La relación *jefe*.

4.7 Consultas recursivas**

Considérese una base de datos que contenga información sobre los empleados de una empresa. Supóngase que se tiene una relación *jefe*(*nombre_empleado*, *nombre jefe*), que especifica qué empleado es supervisado directamente por cada jefe. La Figura 4.11 muestra un ejemplo de la relación *jefe*.

Supóngase ahora que se desea determinar los empleados que son supervisados, directa o indirectamente por un jefe dado—por ejemplo, Santos. Es decir, se desea determinar los empleados supervisados directamente por Santos o por alguien que esté supervisado por Santos, o por alguien que esté supervisado por alguien que esté supervisado por Santos, y así sucesivamente.

Por tanto, si el jefe de Alández es Bariego y el jefe de Bariego es Erice y el jefe de Erice es Santos, entonces Alández, Bariego y Erice son los empleados supervisados por Santos.

4.7.1 Cierre transitivo mediante iteración

Una manera de escribir la consulta anterior es emplear la iteración: En primer lugar hay que determinar las personas que trabajan bajo Santos, luego las que trabajan bajo el primer conjunto, etc. La Figura 4.12 muestra la función *buscaEmpl(jef)* para llevar a cabo esa tarea; la función toma el nombre del jefe como parámetro (*jef*), calcula el conjunto de todos los subordinados directos e indirectos de ese jefe y devuelve el conjunto.

El procedimiento utiliza tres tablas temporales: *empl*, que se utiliza para almacenar el conjunto de tuplas que se va a devolver; *empnuevo*, que almacena los empleados localizados en la iteración anterior; y *temp*, que se utiliza como almacenamiento temporal mientras se manipulan los conjuntos de empleados. El procedimiento inserta todos los empleados que trabajan directamente para *jef* en *empnuevo* antes del bucle **repeat**. El bucle **repeat** añade primero todos los empleados de *empnuevo* a *empl*. Luego calcula los empleados que trabajan para los que están en *empnuevo*, excepto los que ya se sabe que son empleados de *jef* y los almacena en la tabla temporal *temp*. Finalmente, sustituye el contenido de *empnuevo* por el de *temp*. El bucle **repeat** termina cuando no encuentra más subordinados (indirectos) nuevos.

La Figura 4.13 muestra los empleados que se encontrarían en cada iteración si el procedimiento se llamara para el jefe llamado Santos.

Hay que destacar que el uso de la cláusula **except** en la función garantiza que la función trabaje incluso en el caso (anormal) de que haya un ciclo en la jerarquía de jefes. Por ejemplo, si *a* trabaja para *b*, *b* trabaja para *c* y *c* trabaja para *a*, hay un ciclo.

Aunque los ciclos pueden resultar poco realistas en el control de organigramas, son posibles en otras aplicaciones. Por ejemplo, supóngase que se tiene la relación *vuelos(a, desde)* que indica las ciudades a las que se puede llegar desde otra ciudad mediante un vuelo directo. Se puede escribir un código parecido al de la función *buscaEmpl* para determinar todas las ciudades a las que se puede llegar mediante una serie de uno o más vuelos desde una ciudad dada. Todo lo que hay que hacer es sustituir *jefe* por *vuelo* y el nombre de los atributos de manera acorde. En esta situación sí que puede haber ciclos de alcanzabilidad, pero la función trabajaría correctamente, ya que eliminaría las ciudades que ya se hubieran tomado en cuenta.

```

create function buscaEmpl(jef char(10))
    -- Busca todos los empleados que trabajan directa o indirectamente
    -- para jef
returns table (nombre char(10))
    -- La relación jefe(nombre_empleado, nombre_jefe)
    -- especifica quién trabaja directamente para quién.
begin
    create temporary table empl(nombre char(10));
        -- la tabla empl almacena el conjunto de empleados que
        -- se va a devolver
    create temporary table empnuevo(nombre char(10));
        -- la tabla empnuevo contiene los empleados hallados
        -- en la iteración anterior
    create temporary table temp(nombre char(10));
        -- la tabla temp es una tabla temporal utilizada
        -- para almacenar los resultados intermedios
    insert into empnuevo
        select nombre_empleado
        from jefe
        where nombre_jefe = jef
repeat
    insert into empl
        select nombre
        from empnuevo;
    insert into temp
        (select jefe.nombre_empleado
        from empnuevo, jefe
        where empnuevo.nombre_empleado = jefe.nombre_jefe;
    )
    except (
        select nombre_empleado
        from empl
    );
    delete from empnuevo;
    insert into empnuevo
        select *
        from temp;
    delete from temp;
until not exists (select * from empnuevo)
end repeat;
return table empl
end

```

Figura 4.12 Búsqueda de todos los empleados de cada jefe.

4.7.2 Recursión en SQL

El **cierre transitivo** de la relación *jefe* es una relación que contiene todos los pares (*emp*, *jef*) tales que *emp* es subordinado directo o indirecto de *jef*. Hay numerosas aplicaciones que exigen el cálculo de cierres transitivos parecidos sobre las **jerarquías**. Por ejemplo, las organizaciones suelen constar de varios niveles de unidades organizativas. Las máquinas constan de componentes que, a su vez, tienen subcomponentes, etc.; por ejemplo, una bicicleta puede tener subcomponentes como las ruedas y los pedales

Número de iteración	Tuplas de <i>empl</i>
0	
1	(Dalma), (Erice)
2	(Dalma), (Erice), (Bariego), (Corbacho)
3	(Dalma), (Erice), (Bariego), (Corbacho), (Alández)
4	(Dalma), (Erice), (Bariego), (Corbacho), (Alández)

Figura 4.13 Empleados de Santos en las iteraciones de la función *buscaEmpl*.

que, a su vez, tienen subcomponentes como las cámaras, las llantas y los radios. El cierre transitivo puede utilizarse sobre esas jerarquías para determinar, por ejemplo, todos los componentes de la bicicleta.

Resulta muy poco práctico especificar el cierre transitivo empleando la iteración. Hay un enfoque alternativo, el uso de las definiciones recursivas de las vistas, que resulta más sencillo de utilizar.

Se puede emplear la recursión para definir el conjunto de empleados controlados por un jefe determinado, digamos Santos, del modo siguiente. Las personas supervisadas (directa o indirectamente) por Santos son:

1. Personas cuyo jefe es Santos.
2. Personas cuyo jefe está supervisado (directa o indirectamente) por Santos.

Obsérvese que el caso 2 es recursivo, ya que define el conjunto de personas supervisadas por Santos en términos del conjunto de personas supervisadas por Santos. Otros ejemplos de cierre transitivo, como la búsqueda de todos los subcomponentes (directos o indirectos) de un componente dado también puede definirse de manera parecida, recursivamente.

Desde la versión SQL:1999 la norma de SQL soporta una forma limitada de recursión, que emplea la cláusula **with recursive**, en la que las vistas (o las vistas temporales) se expresan en términos de sí mismas. Se pueden utilizar consultas recursivas, por ejemplo, para expresar de manera concisa el cierre transitivo. Recuérdese que la cláusula **with** se emplea para definir una vista temporal cuya definición sólo está disponible para la consulta en la que se define. La palabra clave adicional **recursive** especifica que la vista es recursiva.

Por ejemplo, se pueden buscar todos los pares (*emp*, *jef*) tales que *emp* esté directa o indirectamente mandado por *jef*, empleando la vista recursiva de SQL que aparece en la Figura 4.14.

Todas las vistas recursivas deben definirse como la unión de dos subconsultas: una **consulta básica** que no es recursiva y una **consulta recursiva** que utiliza la vista recursiva. En el ejemplo de la Figura 4.14 la consulta básica es la selección sobre *jefe*, mientras que la consulta recursiva calcula la reunión de *jefe* y *empl*.

El significado de las vistas recursivas se comprende mejor de la manera siguiente. En primer lugar hay que calcular la consulta básica y añadir todas las tuplas resultantes a la vista (que se halla inicialmente vacía). A continuación hay que calcular la consulta recursiva empleando el contenido actual de la vista y volver a añadir todas las tuplas resultantes a la vista. Hay que seguir repitiendo este paso hasta que no

```

with recursive empl(nombre_empleado, nombre_jefe) as (
    select nombre_empleado, nombre_jefe
    from jefe
    union
        select jefe.nombre_empleado, empl.nombre_jefe
        from jefe, empl
        where jefe.nombre_jefe = empl.nombre_empleado
)
select *
from empl

```

Figura 4.14 Consulta recursiva en SQL.

se añada ninguna tupla nueva a la vista. La instancia resultante de la vista se denomina **punto fijo** de la definición recursiva de la vista. (El término “fijo” hace referencia al hecho de que ya no se producen más modificaciones). La vista, por tanto, se define para que contenga exactamente las tuplas de la instancia del punto fijo.

Aplicando la lógica anterior a nuestro ejemplo, se hallan primero todos los subordinados directos de cada jefe ejecutando la consulta básica. La consulta recursiva añade un nivel de empleados en cada iteración, hasta alcanzar la profundidad máxima de la relación jefe-empleado. En ese punto ya no se añaden nuevas tuplas a la vista y la iteración ha alcanzado un punto fijo.

Obsérvese que no se exige que el sistema de bases de datos utilice esta técnica iterativa para calcular el resultado de la consulta recursiva; puede obtener el mismo resultado empleando otras técnicas que pueden ser más eficientes.

Hay algunas restricciones para la consulta recursiva de la vista recursiva, concretamente, la consulta debe ser **monótona**, es decir, su resultado sobre la instancia de una relación vista V_1 debe ser un superconjunto de su resultado sobre la instancia de una relación vista V_2 si V_1 es un superconjunto de V_2 . De manera intuitiva, si se añaden más tuplas a la relación vista, la consulta recursiva debe devolver, como mínimo, el mismo conjunto de tuplas que antes, y posiblemente devuelva tuplas adicionales.

En concreto, las consultas recursivas no deben emplear ninguno de los constructores siguientes, ya que harían que la consulta no fuera monótona:

- Agregación sobre la vista recursiva.
- **not exists** sobre una subconsulta que utiliza la vista recursiva.
- Diferencia de conjuntos (**except**) cuyo lado derecho utilice la vista recursiva.

Por ejemplo, si la consulta recursiva fuera de la forma $r - v$, donde v es la vista recursiva, si se añade una tupla a v , el resultado de la consulta puede hacerse más pequeño; la consulta, por tanto, no es monótona.

El significado de las vistas recursivas puede definirse mediante el procedimiento recursivo mientras la consulta recursiva sea monótona; si la consulta recursiva no es monótona, el significado de la consulta resulta difícil de definir. SQL, por tanto, exige que las consultas sean monótonas. Las consultas recursivas se estudian con más detalle en el contexto del lenguaje de consultas Datalog, en el Apartado 5.4.6.

SQL también permite la creación de vistas permanentes definidas de manera recursiva mediante el uso de **create recursive view**, en lugar de **with recursive**. Algunas implementaciones soportan consultas recursivas que emplean una sintaxis diferente; consúltese el manual del sistema correspondiente para conocer más detalles.

4.8 Características avanzadas de SQL**

El lenguaje SQL ha crecido durante las dos últimas décadas desde ser un lenguaje sencillo con pocas características a ser un lenguaje bastante complejo con características para satisfacer a muchos tipos diferentes de usuarios. En este capítulo ya se han tratado los fundamentos de SQL. En este apartado se presentan al lector algunas de las características más complejas de SQL. Muchas de estas características se han introducido en versiones de la norma de SQL relativamente recientes y puede que no las soporten todos los sistemas de bases de datos.

4.8.1 Creación de tablas a partir de otras

Las aplicaciones necesitan a menudo la creación de tablas que tengan el mismo esquema que otra tabla ya existente. SQL proporciona la extensión **create table like** para soportar esta tarea:

```
create table cuenta_temp like cuenta
```

La instrucción anterior crea una nueva tabla *cuenta_temp* que tiene el mismo esquema que *cuenta*.

Al escribir consultas complejas suele resultar útil almacenar el resultado de la consulta en forma de nueva tabla; esa tabla suele ser temporal. Hacen falta dos instrucciones, una para crear la tabla (con las columnas correspondientes) y otra para insertar el resultado de la consulta en la tabla. SQL:2003

proporciona una técnica más sencilla para crear una tabla que contenga el resultado de una consulta. Por ejemplo, la instrucción siguiente crea la tabla *t1*, que contiene el resultado de una consulta.

```
create table t1 as
  (select *
   from cuenta
   where nombre_sucursal = 'Navacerrada')
with data
```

De manera predeterminada, el nombre y el tipo de datos de las columnas se deduce del resultado de la consulta. Se puede poner nombre a las columnas de manera explícita escribiendo la lista de los nombres de las columnas tras el nombre de la relación. Si se omite la cláusula **with data**, la tabla se crea, pero no se rellena con datos.

La instrucción **create table ... as** anterior se parece mucho a la instrucción **create view**, y las dos se definen mediante consultas. La principal diferencia es que el contenido de la tabla se define al crearla, mientras que el contenido de una vista siempre refleja el resultado actual de la consulta.

Hay que tener en cuenta que varias implementaciones soportan la funcionalidad de **create table ... like** y **create table ... as** pero emplean sintaxis diferentes; consultese el manual del sistema respectivo para obtener más detalles.

4.8.2 Otros aspectos sobre las subconsultas

SQL:2003 permite que las subconsultas se produzcan siempre que haga falta un valor, siempre y cuando la consulta sólo devuelva un valor; estas subconsultas se denominan **subconsultas escalares**. Por ejemplo, se puede utilizar una subconsulta en la cláusula **select** como se ilustra en el ejemplo siguiente, que genera una lista de todos los clientes y del número de cuentas que tienen:

```
select nombre_cliente,
       (select count(*)
        from cuenta
        where cuenta.nombre_cliente = cliente.nombre_cliente) as numero_cuentas
     from cliente
```

Está garantizado que la subconsulta del ejemplo anterior sólo devuelva un único valor, ya que tiene un agregado **count(*)** sin el correspondiente **group by**. Las subconsultas sin agregados también están permitidas. Estas consultas pueden, en teoría, devolver más de una respuesta; si lo hacen, se produce un error en tiempo de ejecución.

Las subconsultas de la cláusula **select** de la consulta exterior pueden tener acceso a los atributos de las relaciones de la cláusula **from** de la consulta exterior, como *cliente.nombre_cliente* en este ejemplo.

Las subconsultas de la cláusula **from** (que ya se ha estudiado en el Apartado 3.8.1), sin embargo, normalmente no pueden tener acceso a los atributos de otras relaciones de la cláusula **from**; SQL:2003 soporta la cláusula **lateral**, que permite que las subconsultas de la cláusula **from** tengan acceso a los atributos de las subconsultas anteriores de esa misma cláusula **from**. Por tanto, la consulta anterior puede escribirse también del modo siguiente

```
select nombre_cliente, numero_cuentas
      from cliente,
            lateral(select count(*)
                    from cuenta
                    where cuenta.nombre_cliente = cliente.nombre_cliente)
              as este_cliente(numero_cuentas)
```

4.8.3 Constructores avanzados para la actualización de las bases de datos

Supóngase que se tiene la relación *fondos_recibidos*(*número_cuenta*, *importe*) que almacena los fondos recibidos (digamos, mediante transferencias electrónicas) para cada cuenta de un grupo. Supóngase ahora

que se desea añadir los importes a los saldos de las cuentas correspondientes. Para utilizar la actualización **update** de SQL para llevar a cabo esta tarea hay que examinar la tabla *fondos_recibidos* para cada tupla de la tabla *cuenta*. Se pueden utilizar subconsultas en la cláusula update para llevar a cabo esta tarea, como se muestra a continuación. En aras de la sencillez se supone que la relación *fondos_recibidos* contiene, como máximo, una tupla por cada cuenta.

```
update cuenta set saldo = saldo +
    (select importe
     from fondos_recibidos
      where fondos_recibidos.número_cuenta = cuenta.número_cuenta)
where exists(  

    select *
     from fondos_recibidos
      where fondos_recibidos.número_cuenta = cuenta.número_cuenta)
```

Téngase en cuenta que la condición de la cláusula **where** de la actualización garantiza que sólo se actualicen las cuentas con sus tuplas correspondientes en *fondos_recibidos*, mientras que la subconsulta de la cláusula **set** calcula el montante que hay que añadir a cada una de esas cuentas.

Hay muchas aplicaciones que necesitan actualizaciones como la que se acaba de ilustrar. Generalmente hay una tabla, que se puede llamar **tabla maestra**, y las actualizaciones de la tabla maestra se reciben por lotes. Luego hay que actualizar a su vez la tabla maestra. SQL:2003 proporciona un constructor especial, denominado **merge**, para simplificar la labor de mezcla de toda esa información. Por ejemplo, la actualización anterior puede expresarse empleando **merge** de la manera siguiente:

```
merge into cuenta as C
using   (select *
            from fondos_recibidos) as F
          on (C.número_cuenta = F.número_cuenta)
when matched then
    update set saldo = saldo+F.importe
```

Cuando un registro de la subconsulta de la cláusula **using** coincide con un registro de la relación *cuenta*, se ejecuta la cláusula **when matched**, que puede ejecutar una actualización de la relación; en este caso, el registro coincidente de la relación *cuenta* se actualiza de la manera mostrada.

La instrucción **merge** también puede tener una cláusula **when not matched then**, que permite la inserción de registros nuevos en la relación. En el ejemplo anterior, cuando no hay cuenta coincidente para alguna tupla de *fondos_recibidos*, la acción de inserción puede crear un nuevo registro de *cuenta* (con un valor nulo para *nombre_sucursal*) utilizando la cláusula siguiente.

```
when not matched then
    insert values (F.número_cuenta, null, F.importe)
```

Aunque no sea muy significativo en este ejemplo², la cláusula **when not matched clause** puede resultar bastante útil en otros casos. Por ejemplo, supóngase que la relación local es copia de una relación maestra y se reciben de la relación maestra registros actualizados y otros recién insertados. La instrucción **merge** puede actualizar los registros coincidentes (serán registros antiguos actualizados) e insertar los registros que no coincidan (serán registros nuevos).

No todas las implementaciones de SQL soportan actualmente la instrucción **merge**; consultese el manual del sistema correspondiente para obtener más detalles.

2. Una solución mejor en este caso habría sido insertar estos registros en una relación de error, pero eso no se puede hacer con la instrucción **merge**.

4.9 Resumen

- El lenguaje de definición de datos de SQL ofrece soporte para la definición de tipos predefinidos como la fecha y la hora, así como para tipos de dominios definidos por los usuarios.
- Las restricciones de dominio especifican el conjunto de valores posibles que pueden asociarse con cada atributo. Estas restricciones también pueden prohibir el uso de valores nulos para atributos concretos.
- Las restricciones de integridad referencial aseguran que un valor que aparece en una relación para un conjunto de atributos dado aparezca también para un conjunto de atributos concreto en otra relación.
- Los assertos son expresiones declarativas que establecen predicados que necesitamos que siempre sean ciertos.
- Cada usuario puede tener varias formas de autorización para diferentes partes de la base de datos. La autorización es un medio de proteger el sistema de bases de datos contra el acceso malintencionado o no autorizado.
- Las consultas SQL se pueden invocar desde lenguajes anfitriones mediante SQL incorporado y SQL dinámico. Las normas ODBC and JDBC definen interfaces para programas de aplicación para el acceso a las bases de datos de SQL desde programas en los lenguajes C y Java. Los programadores utilizan cada vez más estas APIs para el acceso a las bases de datos.
- Las funciones y los procedimientos pueden definirse empleando SQL. También se han descrito las extensiones procedimentales proporcionadas por SQL:1999, que permiten la iteración y las instrucciones condicionales (if-then-else).
- Algunas consultas, como el cierre transitivo, pueden expresarse tanto mediante la iteración como mediante las consultas recursivas de SQL. La recursión puede expresarse mediante las vistas recursivas o mediante cláusulas **with** recursivas.
- También se ha visto una breve introducción de algunas características avanzadas de SQL, que simplifican determinadas tareas relacionadas con la definición de datos y la consulta y la actualización de los datos.

Términos de repaso

- Tipos definidos por los usuarios.
- Dominios.
- Objetos de gran tamaño.
- Catálogos.
- Esquemas.
- Restricciones de integridad.
- Restricciones de dominios.
- Restricción de unicidad.
- Cláusula **check**.
- Integridad referencial.
- Restricción de clave primaria.
- Restricción de clave externa.
- Borrados en cascada.
- Actualizaciones en cascada.
- Asertos.
- Autorizaciones.
- Privilegios:
 - **select**.
 - **insert**.
 - **update**.
 - **delete**.
 - **all privileges**.
- Concesión de privilegios.
- Retirada de privilegios.
- SQL incorporado.
- Cursores.
- Cursores actualizables.
- SQL dinámico.
- ODBC.
- JDBC.
- Instrucciones preparadas.

- Acceso a los metadatos.
- Funciones de SQL.
- Procedimientos almacenados.
- Constructores procedimentales.
- Rutinas de otros lenguajes.
- Consultas recursivas.
- Consultas monótonas.
- Instrucción **merge**.

Ejercicios prácticos

- 4.1** Complétense la definición del LDD de SQL de la base de datos bancaria de la Figura 4.2 para que incluya las relaciones *préstamo* y *prestatario*.
- 4.2** Considérese la siguiente base de datos relacional:

```

empleado(nombre_empleado, calle, ciudad)
trabaja(nombre_empleado, nombre_empresa, sueldo)
empresa(nombre_empresa, ciudad)
jefe(nombre_empleado, nombre_jefe)

```

Dese una definición de esta base de datos en el LDD de SQL. Identifíquense las restricciones de integridad referencial que deben cumplirse e inclúyanse en la definición del LDD.

- 4.3** Escribanse condiciones **check** para el esquema que se ha definido en el Ejercicio 4.2 para garantizar que:
- Cada empleado trabaje para una empresa ubicada en su ciudad de residencia.
 - Ningún empleado gane un sueldo mayor que el de su jefe.
- 4.4** SQL permite que las dependencias de clave externa hagan referencia a la misma relación, como en el ejemplo siguiente:

```

create table jefe
  (nombre_empleado char(20) not null,
   nombre_jefe      char(20) not null,
   primary key nombre_empleado,
   foreign key (nombre_jefe) references jefe
     on delete cascade )

```

Aquí, *nombre_empleado* es clave de la tabla *jefe*, lo que significa que cada empleado tiene como máximo un director. La cláusula de clave externa exige que cada director sea también empleado. Explíquese exactamente lo que ocurre cuando se borra una tupla de la relación *jefe*.

- 4.5** Escribábase un aserto para la base de datos bancaria que garantice que el valor de los activos de la sucursal de Navacerrada sea igual a la suma de todos los importes prestados por esa sucursal.
- 4.6** Describanse las circunstancias bajo las cuales es preferible utilizar SQL incorporado en lugar de SQL solo o un lenguaje de programación de propósito general sin más.

Ejercicios

- 4.7** Las restricciones de integridad referencial tal y como se han definido en este capítulo implican exactamente a dos relaciones. Considérese una base de datos que incluye las relaciones siguientes:

```

trabajador_jornada_completa(nombre, despacho, teléfono, sueldo)
trabajador_tiempo_parcial(nombre, sueldo_por_hora)
dirección(nombre, calle, ciudad)

```

Supóngase que se desea exigir que cada nombre que aparece en *dirección* aparezca también en *trabajador_jornada_completa* o en *trabajador_tiempo_parcial*, pero no necesariamente en ambos.

- a. Propóngase una sintaxis para expresar estas restricciones.
- b. Explíquense las acciones que debe realizar el sistema para aplicar una restricción de este tipo.

- 4.8 Escríbase una función de Java que emplee las características para metadatos de JDBC y tome un *ResultSet* como parámetro de entrada e imprima el resultado en forma tabular, con los nombres correspondientes como encabezados de las columnas.
- 4.9 Escríbase una función de Java que emplee las características para metadatos de JDBC e imprima una lista de todas las relaciones de la base de datos, mostrando para cada relación el nombre y el tipo de datos de los atributos.

4.10 Considérese una base de datos de empleados con dos relaciones

$$\begin{aligned} &\text{empleado}(\underline{\text{nombre_empleado}}, \underline{\text{calle}}, \underline{\text{ciudad}}) \\ &\text{trabaja}(\underline{\text{nombre_empleado}}, \underline{\text{nombre_empresa}}, \underline{\text{sueldo}}) \end{aligned}$$

en las que se han subrayado las claves primarias. Escríbase una consulta para determinar las empresas cuyos empleados ganan sueldos más altos, en media, que el sueldo medio de Banco Importante.

- a. Empleando las funciones de SQL necesarias.
- b. Sin emplear las funciones de SQL.

- 4.11 Reescríbase la consulta del Apartado 4.6.1 que devuelve el nombre, la calle y la ciudad de todos los clientes con más de una cuenta en el banco empleando la cláusula **with** en lugar de la llamada a una función.
- 4.12 Compárese el uso de SQL incrustado con el uso en SQL de funciones definidas en lenguajes de programación de propósito general. ¿En qué circunstancias es preferible utilizar cada una de estas características?

4.13 Modifíquese la consulta recursiva de la Figura 4.14 para definir la relación

$$\text{profundidad_empleado}(\underline{\text{nombre_empleado}}, \underline{\text{nombre_jefe}}, \underline{\text{profundidad}})$$

en la que el atributo *profundidad* indica el número de niveles de jefes intermedios que hay entre el empleado y su jefe. Los empleados que se hallan directamente bajo un jefe tienen una profundidad de 0.

4.14 Considérese el esquema relacional

$$\begin{aligned} &\text{componente}(\underline{\text{id_componente}}, \underline{\text{nombre}}, \underline{\text{coste}}) \\ &\text{subcomponente}(\underline{\text{id_componente}}, \underline{\text{id_subcomponente}}, \underline{\text{recuento}}) \end{aligned}$$

La tupla $(c_1, c_2, 3)$ de la relación *subcomponente* denota que el componente con identificador de componente c_2 es un subcomponente directo del componente con identificador de componente c_1 , y que c_1 tiene 3 copias de c_2 . Téngase en cuenta que c_2 puede, a su vez, tener más subcomponentes. Escríbase una consulta recursiva de SQL que genere el nombre de todos los subcomponentes del componente con identificador “C-100”.

- 4.15 Considérese nuevamente el esquema relacional del Ejercicio 4.14. Escríbase una función de JDBC que utilice SQL no recursivo para determinar el coste total del componente “C-100”, incluido el coste de todos sus subcomponentes. Hay que asegurarse de tener en cuenta el hecho de que cada componente puede tener varios ejemplares de un subcomponente. Se puede utilizar la recursión en Java si se desea.

Notas bibliográficas

Véanse las notas bibliográficas del Capítulo 3 para consultar las referencias a las normas de SQL y a los libros sobre SQL.

Muchos productos de bases de datos soportan más características de SQL que las especificadas en las normas, y puede que no soporten algunas características recogidas en ellas. Se puede conseguir más información sobre estas características en el manual del producto correspondiente.

java.sun.com/docs/books/tutorial es una excelente fuente de información actualizada sobre JDBC, y sobre Java en general. Las referencias a los libros sobre Java (incluido JDBC) también están disponibles en este URL. El API de ODBC se describe en Microsoft [1997] y en Sanders [1998]. Melton y Eisenberg [2000] proporcionan una guía de SQLJ, JDBC y las tecnologías relacionadas. Se puede encontrar más información sobre ODBC, ADO y ADO.NET en msdn.microsoft.com/data.

Otros lenguajes relacionales

En el Capítulo 2 se ha descrito el álgebra relacional que constituye la base de SQL el lenguaje de consultas más usado. SQL se ha tratado con gran detalle en los Capítulos 3 y 4. En este capítulo se estudiarán primero otros dos lenguajes formales, el cálculo relacional de tuplas y el cálculo relacional de dominios, que son lenguajes de consultas declarativos basados en la lógica matemática. Estos dos lenguajes formales constituyen la base de dos lenguajes más fáciles de usar, QBE y Datalog, que se estudian más adelante en este mismo capítulo.

A diferencia de SQL, QBE es un lenguaje gráfico en el que las consultas *parecen* tablas. QBE y sus variantes se usan mucho en sistemas de bases de datos para computadoras personales. Datalog tiene una sintaxis derivada del lenguaje Prolog. Aunque actualmente no se usa de forma comercial, Datalog se ha usado en varios sistemas de bases de datos en investigación.

Para QBE y Datalog se presentan los constructores y los conceptos fundamentales en lugar de complementos manuales de usuario. Hay que tener presente que las diferentes implementaciones de cada lenguaje pueden diferir en los detalles, o sólo dar soporte a un subconjunto del lenguaje completo.

5.1 El cálculo relacional de tuplas

Cuando se escribe una expresión del álgebra relacional se proporciona una serie de procedimientos que generan la respuesta a la consulta. El cálculo relacional de tuplas, en cambio, es un lenguaje de consultas **no procedimental**. Describe la información deseada sin dar un procedimiento concreto para obtenerla.

Las consultas se expresan en el cálculo relacional de tuplas como

$$\{t \mid P(t)\}$$

es decir, es el conjunto de todas las tuplas t tales que el predicado P es cierto para t . Siguiendo la notación usada anteriormente, se usa $t[A]$ para denotar el valor de la tupla t en el atributo A y $t \in r$ para denotar que la tupla t está en la relación r .

Antes de dar una definición formal del cálculo relacional de tuplas se volverán a examinar algunas de las consultas para las que se escribieron expresiones del álgebra relacional en el Apartado 2.2. Recuérdese que las consultas se realizan sobre el esquema siguiente:

```
sucursal(nombre_sucursal, ciudad_sucursal, activos)
cliente (nombre_cliente, calle_cliente, ciudad_cliente)
préstamo (número_préstamo, nombre_sucursal, importe)
prestatario (nombre_cliente, número_préstamo)
cuenta (número_cuenta, nombre_sucursal, saldo)
impositor (nombre_cliente, número_cuenta)
```

5.1.1 Ejemplos de consultas

Determinar el *nombre_sucursal*, *número_préstamo* e *importe* de los préstamos de más de 1.200 €:

$$\{t \mid t \in \text{préstamo} \wedge t[\text{importe}] > 1200\}$$

Supóngase que sólo se desea obtener el atributo *número_préstamo*, en vez de todos los atributos de la relación *préstamo*. Para escribir esta consulta en cálculo relacional de tuplas hay que incluir una expresión de relación sobre el esquema (*número_préstamo*). Se necesitan las tuplas de (*número_préstamo*) tales que hay una tupla de *préstamo* con el atributo *importe* > 1200. Para expresar esta solicitud hay que usar el constructor “existe” de la lógica matemática. La notación

$$\exists t \in r (Q(t))$$

significa “existe una tupla *t* de la relación *r* tal que el predicado *Q(t)* es cierto”.

Usando esta notación se puede escribir la consulta “Determinar el número de préstamo de todos los préstamos con importe superior a 1.200 €” como:

$$\begin{aligned} \{t \mid \exists s \in \text{préstamo} (t[\text{número_préstamo}] = s[\text{número_préstamo}] \\ \wedge s[\text{importe}] > 1200)\} \end{aligned}$$

En español la expresión anterior se lee “el conjunto de todas las tuplas *t* tales que existe una tupla *s* de la relación *préstamo* para la que los valores de *t* y de *s* para el atributo *número_préstamo* son iguales y el valor de *s* para el atributo *importe* es mayor que 1.200 €”.

La variable tupla *t* sólo se define para el atributo *número_préstamo*, dado que es el único atributo para el que se especifica una condición para *t*. Por tanto, el resultado es una relación de (*número_préstamo*).

Considérese la consulta “Determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada”. Esta consulta es un poco más compleja que las anteriores porque implica a dos relaciones: *prestatario* y *préstamo*. Como se verá, sin embargo, todo lo que necesita es que tengamos dos cláusulas “existe” en la expresión del cálculo relacional de tuplas, relacionadas mediante *y* (\wedge). La consulta se escribe de la manera siguiente:

$$\begin{aligned} \{t \mid \exists s \in \text{prestatario} (t[\text{nombre_cliente}] = s[\text{nombre_cliente}] \\ \wedge \exists u \in \text{préstamo} (u[\text{número_préstamo}] = s[\text{número_préstamo}] \\ \wedge u[\text{nombre_sucursal}] = \text{"Navacerrada"}))\} \end{aligned}$$

En español esta expresión es “el conjunto de todas las tuplas (*nombre_cliente*) tales que el cliente tiene un préstamo concedido en la sucursal de Navacerrada”. La variable tupla *u* garantiza que el cliente sea prestatario de la sucursal de Navacerrada. La variable tupla *s* está restringida para que corresponda al mismo número de préstamo que *s*. El resultado de esta consulta se muestra en la Figura 5.1.

Para determinar todos los clientes del banco que tienen concedido un préstamo, abierta una cuenta o ambas cosas, se usó la operación unión del álgebra relacional. En el cálculo relacional de tuplas serán necesarias dos instrucciones “existe” conectadas mediante *o* (\vee):

$$\begin{aligned} \{t \mid \exists s \in \text{prestatario} (t[\text{nombre_cliente}] = s[\text{nombre_cliente}]) \\ \vee \exists u \in \text{impositor} (t[\text{nombre_cliente}] = u[\text{nombre_cliente}])\} \end{aligned}$$

Esta expresión da el conjunto de todas las tuplas de *nombre_cliente* tales que se cumple al menos una de las condiciones siguientes:

nombre_cliente
Fernández
López

Figura 5.1 Nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada.

- *nombre_cliente* aparece en alguna tupla de la relación *prestatario* como prestatario del banco.
- *nombre_cliente* aparece en alguna tupla de la relación *impositor* como impositor del banco.

Si algún cliente tiene concedido un préstamo y abierta una cuenta en el banco, ese cliente sólo aparece una vez en el resultado, ya que la definición matemática de conjunto no permite elementos duplicados. El resultado de esta consulta ya se mostró en la Figura 2.11.

Si sólo queremos conocer los clientes que tienen en el banco tanto una cuenta como un préstamo, todo lo que hay que hacer es cambiar en la expresión anterior la *o* (\vee) por una *y* (\wedge) en la expresión anterior.

$$\{t \mid \exists s \in \text{prestatario} (t[\text{nombre_cliente}] = s[\text{nombre_cliente}]) \wedge \exists u \in \text{impositor} (t[\text{nombre_cliente}] = u[\text{nombre_cliente}])\}$$

El resultado de esta consulta se mostró en la Figura 2.19.

Considérese ahora la consulta “Determinar todos los clientes que tienen una cuenta abierta en el banco pero no tienen concedido ningún préstamo”. La expresión del cálculo relacional de tuplas para esta consulta es parecida a las que se acaban de ver, salvo en el uso del símbolo *no* (\neg):

$$\{t \mid \exists u \in \text{impositor} (t[\text{nombre_cliente}] = u[\text{nombre_cliente}]) \wedge \neg \exists s \in \text{prestatario} (t[\text{nombre_cliente}] = s[\text{nombre_cliente}])\}$$

Esta expresión del cálculo relacional de tuplas usa la cláusula $\exists u \in \text{impositor} (\dots)$ para exigir que el cliente tenga una cuenta abierta en el banco, y la cláusula $\neg \exists s \in \text{prestatario} (\dots)$ para eliminar los clientes que aparecen en alguna tupla de la relación *prestatario* por tener un préstamo del banco. El resultado de esta consulta apareció en la Figura 2.12.

La consulta que se tomará ahora en consideración usa la implicación, denotada por \Rightarrow . La fórmula $P \Rightarrow Q$ significa “ P implica Q ”; es decir, “si P es cierto, entonces Q tiene que serlo también”. Obsérvese que $P \Rightarrow Q$ es equivalente lógicamente a $\neg P \vee Q$. El empleo de la implicación en lugar de *no* y *o* sugiere una interpretación más intuitiva de la consulta en español.

Considérese la consulta que se usó en el Apartado 2.3.3 para ilustrar la operación división: “Determinar todos los clientes que tienen una cuenta en todas las sucursales de Arganzuela”. Para escribir esta consulta en el cálculo relacional de tuplas se introduce el constructor “para todo”, denotado por \forall . La notación

$$\forall t \in r (Q(t))$$

significa “ Q es verdadera para todas las tuplas t de la relación r ”.

La expresión para la consulta se escribe de la manera siguiente:

$$\begin{aligned} & \{t \mid \exists r \in \text{cliente} (r[\text{nombre_cliente}] = t[\text{nombre_cliente}]) \wedge \\ & \quad (\forall u \in \text{sucursal} (u[\text{ciudad_sucursal}] = \text{"Arganzuela"} \Rightarrow \\ & \quad \exists s \in \text{impositor} (t[\text{nombre_cliente}] = s[\text{nombre_cliente}]) \\ & \quad \wedge \exists w \in \text{cuenta} (w[\text{número_cuenta}] = s[\text{número_cuenta}]) \\ & \quad \wedge w[\text{nombre_sucursal}] = u[\text{nombre_sucursal}])))\} \end{aligned}$$

En español esta expresión se interpreta como “el conjunto de todos los clientes (es decir, las tuplas t (nombre_cliente)) tales que, para *todas* las tuplas u de la relación *sucursal*, si el valor de u en el atributo *ciudad_sucursal* es Arganzuela, el cliente tiene una cuenta en la sucursal cuyo nombre aparece en el atributo *nombre_sucursal* de u ”.

Obsérvese que hay cierta sutileza en la consulta anterior: si no hay ninguna sucursal en Arganzuela, todos los nombres de cliente satisfacen la condición. La primera línea de la expresión de consulta es crítica en este caso—sin la condición

$$\exists r \in \text{cliente} (r[\text{nombre_cliente}] = t[\text{nombre_cliente}])$$

si no hay ninguna sucursal en Arganzuela, cualquier valor de t (incluyendo los que no son nombres de cliente de la relación *cliente*) valdría.

5.1.2 Definición formal

Con la preparación necesaria adquirida hasta el momento es posible proporcionar una definición formal. Las expresiones del cálculo relacional de tuplas son de la forma:

$$\{t \mid P(t)\}$$

donde P es una *fórmula*. En una fórmula pueden aparecer varias variables tupla. Se dice que una variable tupla es una *variable libre* a menos que esté cuantificada mediante \exists o \forall . Por tanto, en

$$t \in \text{préstamo} \wedge \exists s \in \text{cliente}(t[\text{nombre_sucursal}] = s[\text{nombre_sucursal}])$$

t es una variable libre. La variable tupla s se denomina variable *ligada*.

Las fórmulas del cálculo relacional de tuplas se construyen con *átomos*. Los átomos tienen una de las formas siguientes:

- $s \in r$, donde s es una variable tupla y r es una relación (no se permite el uso del operador \notin)
- $s[x] \Theta u[y]$, donde s y u son variables tuplas, x es un atributo en el que está definida s , y es un atributo en el que está definida u , Θ es un operador de comparación ($<$, \leq , $=$, \neq , $>$, \geq); se exige que los atributos x e y tengan dominios cuyos miembros puedan compararse mediante Θ
- $s[x] \Theta c$, donde s es una variable tupla, x es un atributo en el que está definida s , Θ es un operador de comparación y c es una constante en el dominio del atributo x

Las fórmulas se construyen a partir de los átomos usando las reglas siguientes:

- Cada átomo es una fórmula.
- Si P_1 es una fórmula, también lo son $\neg P_1$ y (P_1) .
- Si P_1 y P_2 son fórmulas, también lo son $P_1 \vee P_2$, $P_1 \wedge P_2$ y $P_1 \Rightarrow P_2$.
- Si $P_1(s)$ es una fórmula que contiene la variable tupla libre s , y r es una relación,

$$\exists s \in r (P_1(s)) \text{ y } \forall s \in r (P_1(s))$$

también son fórmulas.

Igual que en el álgebra relacional, se pueden escribir expresiones equivalentes que no sean idénticas en apariencia. En el cálculo relacional de tuplas estas equivalencias incluyen las tres reglas siguientes:

1. $P_1 \wedge P_2$ es equivalente a $\neg(\neg(P_1) \vee \neg(P_2))$.
2. $\forall t \in r (P_1(t))$ es equivalente a $\neg \exists t \in r (\neg P_1(t))$.
3. $P_1 \Rightarrow P_2$ es equivalente a $\neg(P_1) \vee P_2$.

5.1.3 Seguridad de las expresiones

Queda un último asunto por tratar. Las expresiones del cálculo relacional de tuplas pueden generar relaciones infinitas. Supóngase que se escribe la expresión

$$\{t \mid \neg(t \in \text{préstamo})\}$$

Hay infinitas tuplas que no están en *préstamo*. La mayor parte de estas tuplas contienen valores que ni siquiera aparecen en la base de datos. Resulta evidente que no se desea permitir expresiones de ese tipo.

Para ayudar a definir las restricciones del cálculo relacional de tuplas se introduce el concepto de **dominio** de las fórmulas relacionales de tuplas, P . De manera intuitiva, el dominio de P , denotado por $\text{dom}(P)$, es el conjunto de todos los valores a los que P hace referencia. Esto incluye a los valores mencionados en la propia P , así como a los valores que aparezcan en tuplas de relaciones mencionadas por P . Por tanto, el dominio de P es el conjunto de todos los valores que aparecen explícitamente en P , o en una o más relaciones cuyos nombres aparezcan en P . Por ejemplo, $\text{dom}(t \in \text{préstamo} \wedge t[\text{importe}] > 1200)$ es el conjunto que contiene a 1200 y el conjunto de todos los valores que aparecen en *préstamo*.

Además, $\text{dom}(\neg(t \in \text{préstamo}))$ es el conjunto de todos los valores que aparecen en *préstamo*, dado que la relación *préstamo* se menciona en la expresión.

Se dice que una expresión $\{t \mid P(t)\}$ es *segura* si todos los valores que aparecen en el resultado son valores de $\text{dom}(P)$. La expresión $\{t \mid \neg(t \in \text{préstamo})\}$ no es segura. Obsérvese que $\text{dom}(\neg(t \in \text{préstamo}))$ es el conjunto de todos los valores que aparecen en *préstamo*. Sin embargo, es posible tener una tupla t que no esté en *préstamo* que contenga valores que no aparezcan en *préstamo*. El resto de ejemplos de expresiones del cálculo relacional de tuplas que se han escrito en este apartado son seguros.

5.1.4 Potencia expresiva de los lenguajes

El cálculo relacional de tuplas restringido a expresiones seguras es equivalente en potencia expresiva al álgebra relacional básica (con los operadores \cup , $-$, \times , σ y ρ , pero sin los operadores relacionales extendidos como la proyección generalizada \mathcal{G} y las operaciones de reunión externa). Por tanto, para cada expresión del álgebra relacional que sólo utilice los operadores básicos, existe una expresión equivalente del cálculo relacional de tuplas y viceversa. No se probará aquí esta afirmación; las notas bibliográficas contienen referencias a su demostración. Algunas partes de la misma se incluyen en los ejercicios. Hay que destacar que el cálculo relacional de tuplas no tiene ningún equivalente de la operación agregación, pero se puede extender para contenerla. La extensión del cálculo relacional de tuplas para que maneje las expresiones aritméticas es sencilla.

5.2 El cálculo relacional de dominios

Una segunda forma de cálculo relacional, denominada **cálculo relacional de dominios**, usa variables de *dominio*, que toman sus valores del dominio de un atributo, en vez de hacerlo para una tupla completa. El cálculo relacional de dominios, no obstante, se halla estrechamente relacionado con el cálculo relacional de tuplas.

5.2.1 Definición formal

Las expresiones del cálculo relacional de dominios son de la forma

$$\{< x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n)\}$$

donde x_1, x_2, \dots, x_n representan las variables de dominio, P representa una fórmula compuesta por átomos, como era el caso en el cálculo relacional de tuplas. Los átomos del cálculo relacional de dominios tienen una de las formas siguientes:

- $< x_1, x_2, \dots, x_n > \in r$, donde r es una relación con n atributos y x_1, x_2, \dots, x_n son variables de dominio o constantes de dominio.
- $x \Theta y$, donde x e y son variables de dominio y Θ es un operador de comparación ($<$, \leq , $=$, \neq , $>$, \geq). Se exige que los atributos x e y tengan dominios que puedan compararse mediante Θ .
- $x \Theta c$, donde x es una variable de dominio, Θ es un operador de comparación y c es una constante del dominio del atributo para el que x es una variable de dominio.

Las fórmulas se construyen a partir de los átomos usando las reglas siguientes:

- Los átomos son fórmulas.
- Si P_1 es una fórmula, también lo son $\neg P_1$ y (P_1) .
- Si P_1 y P_2 son fórmulas, también lo son $P_1 \vee P_2$, $P_1 \wedge P_2$ y $P_1 \Rightarrow P_2$.
- Si $P_1(x)$ es una fórmula en x , donde x es una variable libre de dominio, también son fórmulas:

$$\exists x (P_1(x)) \text{ y } \forall x (P_1(x))$$

Como notación abreviada se escribe $\exists a, b, c (P(a, b, c))$ en lugar de $\exists a (\exists b (\exists c (P(a, b, c))))$.

5.2.2 Ejemplos de consultas

Ahora se van a presentar consultas del cálculo relacional de dominios para los ejemplos considerados anteriormente. Obsérvese la similitud de estas expresiones con las expresiones correspondientes del cálculo relacional de tuplas.

- Determinar el número de préstamo, el nombre de la sucursal y el importe de los préstamos de más de 1.200 €:

$$\{< p, s, i > \mid < p, s, i > \in \text{préstamo} \wedge i > 1200\}$$

- Determinar todos los números de préstamo de los préstamos por importe superior a 1.200 €:

$$\{< p > \mid \exists s, i (< p, s, i > \in \text{préstamo} \wedge i > 1200)\}$$

Aunque la segunda consulta tenga un aspecto muy parecido al de la que se escribió para el cálculo relacional de tuplas, hay una diferencia importante. En el cálculo de tuplas, cuando se escribe $\exists s$ para alguna variable tupla s , se vincula inmediatamente con una relación escribiendo $\exists s \in r$. Sin embargo, cuando se usa $\exists s$ en el cálculo de dominios, b no se refiere a una tupla, sino a un valor de dominio. Por tanto, el dominio de la variable s no está restringido hasta que la subfórmula $< p, s, i > \in \text{préstamo}$ restringe s a los nombres de sucursal que aparecen en la relación *préstamo*.

Ahora se mostrarán varios ejemplos de consultas del cálculo relacional de dominios.

- Determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada y determinar el importe del préstamo:

$$\{< n, i > \mid \exists p (< n, p > \in \text{prestatario} \wedge \exists s (< p, s, i > \in \text{préstamo} \wedge s = \text{"Navacerrada"}))\}$$

- Determinar el nombre de todos los clientes que tienen concedido un préstamo, abierta una cuenta o ambas cosas en la sucursal de Navacerrada:

$$\begin{aligned} \{< n > \mid \exists p (&< n, p > \in \text{prestatario} \\ &\wedge \exists s, i (< p, s, i > \in \text{préstamo} \wedge s = \text{"Navacerrada"}) \\ &\vee \exists c (< n, c > \in \text{impositor} \\ &\wedge \exists s, x (< c, s, x > \in \text{cuenta} \wedge s = \text{"Navacerrada"}))\} \end{aligned}$$

- Determinar el nombre de todos los clientes que tienen una cuenta abierta en todas las sucursales situadas en Arganzuela:

$$\begin{aligned} \{< n > \mid \exists d, c (&< n, d, c > \in \text{cliente}) \wedge \\ &\forall x, y, z (< x, y, z > \in \text{sucursal} \wedge y = \text{"Arganzuela"} \Rightarrow \\ &\exists a, b (< a, x, b > \in \text{cuenta} \wedge < n, a > \in \text{impositor}))\} \end{aligned}$$

En español la expresión anterior se interpreta como “el conjunto de todas las tuplas c (*nombre_cliente*) tales que, para todas las tuplas x, y, z (*nombre_sucursal*, *ciudad_sucursal*, *activos*), si la ciudad de la sucursal es Arganzuela, las siguientes afirmaciones son verdaderas:

- Existe una tupla de la relación *cuenta* con número de cuenta a y nombre de sucursal x .
- Existe una tupla de la relación *impositor* con cliente n y número de cuenta a .

5.2.3 Seguridad de las expresiones

Ya se observó que, en el cálculo relacional de tuplas (Apartado 5.1), es posible escribir expresiones que generen relaciones infinitas. Esto llevó a definir la *seguridad* de las expresiones de cálculo relacional de tuplas. Se produce una situación parecida en el cálculo relacional de dominios. Las expresiones como:

$$\{< p, s, i > \mid \neg(< p, s, i > \in \text{préstamo})\}$$

no son seguras porque permiten valores del resultado que no están en el dominio de la expresión.

En el cálculo relacional de dominios también hay que tener en cuenta la forma de las fórmulas dentro de las instrucciones “existe” y “para todo”. Considérese la expresión:

$$\{< x > \mid \exists y (< x, y > \in r) \wedge \exists z (\neg(< x, z > \in r) \wedge P(x, z))\}$$

donde P es una fórmula que implica a x y a z . Se puede comprobar la primera parte de la fórmula, $\exists y (< x, y > \in r)$, tomando en consideración sólo los valores de r . Sin embargo, para comprobar la segunda parte de la fórmula, $\exists z (\neg(< x, z > \in r) \wedge P(x, z))$, hay que tomar en consideración valores de z que no aparecen en r . Dado que todas las relaciones son finitas, no aparece en r un número infinito de valores. Por tanto, no resulta posible en general comprobar la segunda parte de la fórmula sin tomar en consideración un número infinito de valores posibles de z . En lugar de ello se añaden restricciones para prohibir expresiones como la anterior.

En el cálculo relacional de tuplas se restringió cualquier variable cuantificada existencialmente a variar sobre una relación concreta. Dado que no se hizo así en el cálculo de dominios, se añaden reglas a la definición de seguridad para tratar casos como el del ejemplo. Se dice que la expresión:

$$\{< x_1, x_2, \dots, x_n > \mid P(x_1, x_2, \dots, x_n)\}$$

es segura si se cumplen todas las condiciones siguientes:

1. Todos los valores que aparecen en las tuplas de la expresión son valores de $\text{dom}(P)$.
2. Para cada subfórmula “existe” de la forma $\exists x (P_1(x))$, la subfórmula es cierta si y sólo si hay un valor x de $\text{dom}(P_1)$ tal que $P_1(x)$ es cierto.
3. Para cada subfórmula “para todo” de la forma $\forall x (P_1(x))$, la subfórmula es verdadera si y sólo si $P_1(x)$ es cierto para todos los valores x de $\text{dom}(P_1)$.

El propósito de las reglas adicionales es garantizar que se puedan comprobar las subfórmulas “para todo” y “existe” sin tener que comprobar infinitas posibilidades. Considérese la segunda regla de la definición de seguridad. Para que $\exists x (P_1(x))$ sea cierto sólo hay que encontrar una x para la que $P_1(x)$ lo sea. En general habría que comprobar infinitos valores. Sin embargo, si la expresión es segura, se sabe que se puede restringir la atención a los valores de $\text{dom}(P_1)$. Esta restricción reduce las tuplas que hay que tomar en consideración a un número finito.

La situación de las subfórmulas de la forma $\forall x (P_1(x))$ es parecida. Para asegurar que $\forall x (P_1(x))$ es cierto hay que comprobar en general todos los valores posibles, por lo que es necesario examinar infinitos valores. Como antes, si se sabe que la expresión es segura, basta con comprobar $P_1(x)$ para los valores tomados de $\text{dom}(P_1)$.

Todas las expresiones del cálculo relacional de dominios que se han incluido en los ejemplos de consultas de este apartado son seguras.

5.2.4 Potencia expresiva de los lenguajes

Cuando el cálculo relacional de dominios se restringe a las expresiones seguras, es equivalente en potencia expresiva al cálculo relacional de tuplas restringido también a las expresiones seguras. Dado que ya se observó anteriormente que el cálculo relacional de tuplas restringido es equivalente al álgebra relacional, los tres lenguajes siguientes son equivalentes:

- El álgebra relacional básica (sin las operaciones extendidas del álgebra relacional).
- El cálculo relacional de tuplas restringido a las expresiones seguras.
- El cálculo relacional de dominios restringido a las expresiones seguras.

Hay que tener en cuenta que el cálculo relacional de dominios tampoco tiene equivalente para la operación agregación, pero se puede extender fácilmente para contenerla.

5.3 Query-by-Example

Query-by-Example (QBE, Consulta mediante ejemplos) es el nombre tanto de un lenguaje de manipulación de datos como de un sistema de base de datos que incluyó ese lenguaje.

El lenguaje de manipulación de datos QBE tiene dos características distintivas:

1. A diferencia de la mayor parte de los lenguajes de consultas y de programación, QBE presenta una **sintaxis bidimensional**. Las consultas *parecen* tablas. Las consultas de los lenguajes unidimensionales (como SQL) se *pueden* formular en una línea (posiblemente larga). Los lenguajes bidimensionales *necesitan* dos dimensiones para su formulación. (Existe una versión unidimensional de QBE, pero no se considerará en este estudio).
2. Las consultas en QBE se expresan “mediante ejemplos”. En lugar de incluir un procedimiento para obtener la respuesta deseada, se emplea un ejemplo de lo que se desea. El sistema generaliza ese ejemplo para calcular la respuesta a la consulta.

A pesar de estas características tan poco comunes existe una estrecha correspondencia entre QBE y el cálculo relacional de dominios.

Existen dos enfoques de QBE: la versión original basada en texto y una versión gráfica desarrollada posteriormente que soporta el sistema de bases de datos Microsoft Access. En este apartado se ofrece una breve introducción a las características de manipulación de datos de ambas versiones de QBE. En primer lugar se tratarán las características de QBE basado en texto que se corresponden con la cláusula select-from-where de SQL sin agregación ni actualizaciones. Consultense las notas bibliográficas para ver las referencias de las que se puede conseguir más información sobre la manera en que la versión basada en texto de QBE maneja la ordenación de los resultados, la agregación y la actualización. Más adelante, en el Apartado 5.3.6 se tratarán brevemente las características de la versión gráfica de QBE.

5.3.1 Esqueletos de tablas

Una consulta en QBE se expresa mediante **esqueletos de tablas**. Estas tablas presentan el esquema de la relación, como se muestra en la Figura 5.2. En lugar de llenar la pantalla con todos los esqueletos, el usuario elige los que necesita para una determinada consulta y rellena esos esqueletos con **filas de ejemplo**. Cada fila de ejemplo consiste en constantes y *elementos ejemplo*, que son variables de dominio. Para evitar confusiones entre los dos, QBE usa un carácter de subrayado (_) antes de las variables de dominio, como en x, y permite que las constantes aparezcan sin ninguna indicación particular. Este convenio contrasta con el de la mayoría de los lenguajes, en los que las constantes se encierran entre comillas y las variables aparecen sin ninguna indicación.

5.3.2 Consultas sobre una relación

Recuperando el ejemplo bancario habitual, para determinar todos los números de préstamo de la sucursal de Navacerrada se usa el esqueleto de la relación *préstamo* y se rellena del modo siguiente:

préstamo	número_préstamo	nombre_sucursal	importe
P. <u>x</u>		Navacerrada	

Esta consulta indica al sistema que busque tuplas de *préstamo* que tengan “Navacerrada” como valor del atributo *nombre_sucursal*. Para cada tupla de este tipo, el sistema asigna el valor del atributo *número_préstamo* a la variable x. El valor de la variable x se “imprime” (normalmente en pantalla), debido a que el comando P. (acrónimo de *print*—imprimir en inglés) aparece en la columna *número_préstamo* junto a la variable x. Obsérvese que este resultado es parecido al que se obtendría como respuesta a la siguiente consulta del cálculo relacional de dominios:

$$\{\langle x \rangle | \exists s, c (\langle x, s, c \rangle \in \text{préstamo} \wedge s = \text{"Navacerrada"})\}$$

sucursal	nombre_sucursal	ciudad_sucursal	activos

cliente	nombre_cliente	calle_cliente	ciudad_cliente

préstamo	número_préstamo	nombre_sucursal	importe

prestatario	nombre_cliente	número_préstamo

cuenta	número_cuenta	nombre_sucursal	saldo

impositor	nombre_cliente	número_cuenta

Figura 5.2 Esqueletos de tablas de QBE para el ejemplo bancario.

QBE asume que cada posición vacía de una fila contiene una variable única. En consecuencia, si alguna variable no aparece más de una vez en la consulta, se puede omitir. La consulta anterior, por tanto, puede volver a escribirse como:

préstamo	número_préstamo	nombre_sucursal	importe
P.		Navacerrada	

QBE (a diferencia de SQL) realiza de manera automática la eliminación de duplicados. Para evitar la eliminación es necesario insertar el comando ALL. después del comando P.:

préstamo	número_préstamo	nombre_sucursal	importe
P.ALL.		Navacerrada	

Para mostrar la relación *préstamo* completa, se puede crear una única fila con el comando P. en todos los campos. También se puede usar una notación más concisa consistente en colocar una única orden P. en la columna encabezada por el nombre de la relación:

préstamo	número_préstamo	nombre_sucursal	importe
P.			

QBE permite formular consultas que conlleven comparaciones aritméticas (por ejemplo, $>$), en lugar de las comparaciones de igualdad. Por ejemplo, “Determinar el número de todos los préstamos de aquellos préstamos de importe superior a 700 €”:

<i>préstamo</i>	<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>
	P.		>700

Las comparaciones sólo pueden contener una expresión aritmética en el lado derecho de la operación de comparación (por ejemplo, $> (_x + _y - 20)$). La expresión puede contener tanto variables como constantes. El lado izquierdo de la comparación debe estar vacío. Las operaciones aritméticas que son compatibles con QBE son $=, <, \leq, >, \geq, y -$.

Obsérvese que exigir que el lado izquierdo esté vacío implica que no se pueden comparar dos variables con nombres distintos. Esta dificultad se tratará en breve.

Como ejemplo adicional, considérese la consulta “Determinar el nombre de todas las sucursales que no se hallan en Arganzuela”. Esta consulta se puede formular del siguiente modo:

<i>sucursal</i>	<i>nombre_sucursal</i>	<i>ciudad_sucursal</i>	<i>activos</i>
	P.	¬Arganzuela	

El objetivo principal de las variables en QBE es obligar a ciertas tuplas a tener el mismo valor en algunos atributos. Considerese la consulta “Determinar el número de préstamo de todos los préstamos solicitados conjuntamente por Santos y Gómez”:

<i>prestatario</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
	Santos	P. $_x$
	Gómez	$_x$

Para ejecutar esta consulta el sistema busca todos los pares de tuplas de la relación *prestatario*, que coinciden en el atributo *número_préstamo*, para los que el valor del atributo *nombre_cliente* es “Santos” para una tupla y “Gómez” para la otra. A continuación, el sistema muestra el valor del atributo *número_préstamo*.

En el cálculo relacional de dominios la consulta se podría escribir como:

$$\{ \langle p \rangle \mid \exists x (\langle x, p \rangle \in \text{prestatario} \wedge x = \text{"Santos"} \\ \wedge \exists x (\langle x, p \rangle \in \text{prestatario} \wedge x = \text{"Gómez"}) \}$$

Como ejemplo adicional, considérese la consulta “Determinar el nombre de todos los clientes que viven en la misma ciudad que Santos”:

<i>cliente</i>	<i>nombre_cliente</i>	<i>calle_cliente</i>	<i>ciudad_cliente</i>
	P. $_x$ Santos		¬y ¬y

5.3.3 Consultas sobre varias relaciones

QBE permite formular consultas que abarquen varias relaciones diferentes (de igual forma que el producto cartesiano o la reunión natural en el álgebra relacional). Las conexiones entre las diversas relaciones se llevan a cabo a través de variables que obligan a ciertas tuplas a tomar el mismo valor en determinados atributos. Como ejemplo, supóngase que se desea determinar el nombre de todos los clientes que tienen concedido un préstamo en la sucursal de Navacerrada. Esta consulta se puede escribir como:

<i>préstamo</i>	<i>número_préstamo</i>	<i>nombre_sucursal</i>	<i>importe</i>
	$_x$	Navacerrada	
<i>prestatario</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>	
	P. $_y$	$_x$	

Para evaluar la consulta anterior, el sistema busca las tuplas de la relación *préstamo* con “Navacerrada” como valor del atributo *nombre_sucursal*. Para cada una de esas tuplas, el sistema busca las tuplas de la

relación *prestatario* con el mismo valor para el atributo *número_préstamo* que las tuplas de la relación *préstamo*. El sistema muestra el valor del atributo *nombre_cliente*.

Se puede usar una técnica parecida a la anterior para escribir la consulta “Determinar el nombre de todos los clientes que tienen tanto una cuenta abierta como un préstamo concedido en el banco”:

<i>impositor</i>	<i>nombre_cliente</i>	<i>número_cuenta</i>
	P. <i>x</i>	
<i>prestatario</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
	<i>x</i>	

Considérese ahora la consulta “Determinar el nombre de todos los clientes que tienen una cuenta en el banco pero que no tienen concedido ningún préstamo”. En QBE las consultas que expresan negación se expresan con un signo **not** (\neg) debajo del nombre de la relación y junto a una fila de ejemplo:

<i>impositor</i>	<i>nombre_cliente</i>	<i>número_cuenta</i>
	P. <i>x</i>	
<i>prestatario</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
\neg	<i>x</i>	

Compárese la consulta anterior con la formulada anteriormente: “Determinar el nombre de todos los clientes que tienen tanto una cuenta abierta como un préstamo concedido en el banco”. La única diferencia es la aparición del símbolo \neg junto a la fila de ejemplo del esqueleto de la tabla *prestatario*. Esta diferencia, no obstante, tiene un efecto importante en el procesamiento de la consulta. QBE busca todos los valores de *x* para los cuales:

1. Existe una tupla de la relación *impositor* cuyo *nombre_cliente* es la variable de dominio *x*.
2. No existe ninguna tupla de la relación *prestatario* cuyo *nombre_cliente* sea como el de la variable de dominio *x*.

El símbolo \neg se puede leer como “no existe”.

El hecho de que se haya colocado \neg bajo el nombre de la relación, en lugar de hacerlo bajo el nombre de algún atributo es importante. El símbolo \neg bajo el nombre de un atributo es una abreviatura de \neq . Por tanto, para encontrar todos los clientes que tienen al menos dos cuentas se escribe:

<i>impositor</i>	<i>nombre_cliente</i>	<i>número_cuenta</i>
	P. <i>x</i>	\neg <i>y</i>
	<i>x</i>	\neg <i>y</i>

La consulta anterior se lee en español “Mostrar todos los valores de *nombre_cliente* que aparecen al menos en dos tuplas, de las cuales la segunda tiene un *número_cuenta* diferente de la primera”.

5.3.4 Cuadro de condiciones

Algunas veces resulta poco conveniente o imposible expresar todas las restricciones de las variables de dominio dentro de los esqueletos de tablas. Para solucionar este problema, QBE incluye la característica **cuadro de condiciones**, que permite expresar restricciones generales sobre cualquiera de las variables de dominio. QBE permite que aparezcan expresiones lógicas en los cuadros de condiciones. Los operadores lógicos son las palabras **and** (y) y **or** (o), o bien los símbolos “&” y “|”.

Por ejemplo, la consulta “Determinar el número de préstamo de todos los préstamos concedidos a Santos o a Gómez (o a ambos conjuntamente)” se puede escribir como:

<i>prestatario</i>	<i>nombre_cliente</i>	<i>número_préstamo</i>
	$_n$	P. $_x$

<i>condiciones</i>
$_n = \text{Santos} \text{ or } _n = \text{Gómez}$

Es posible expresar esta consulta sin usar un cuadro de condiciones, usando P. en varias filas. Sin embargo, las consultas con P. en varias filas a veces resultan difíciles de entender y es mejor evitarlas.

Como ejemplo adicional, supóngase que se modifica la última consulta del Apartado 5.3.3 para que quede como: “Determinar todos los clientes cuyo nombre no sea ‘Santos’ y que tengan abiertas, al menos, dos cuentas”. Se desea incluir en esta consulta la restricción “ $x \neq \text{Santos}$ ”. Se consigue empleando el cuadro de condiciones e introduciéndole la restricción “ $x \neq \text{Santos}$ ”:

<i>condiciones</i>
$x \neq \text{Santos}$

Cambiando de ejemplo, para determinar todos los números de cuenta con saldos entre 1.300 € y 1.500 €, se escribe:

<i>cuenta</i>	<i>número_cuenta</i>	<i>nombre_sucursal</i>	<i>saldo</i>
	P.		$_x$

<i>condiciones</i>
$_x \geq 1300$
$_x \leq 1500$

Considérese también la consulta “Determinar todas las sucursales con activos superiores a los activos de, al menos, una sucursal con sede en Arganzuela”. Esta consulta se puede escribir como:

<i>sucursal</i>	<i>nombre_sucursal</i>	<i>ciudad_sucursal</i>	<i>activos</i>
	P. $_x$	Arganzuela	$_y$ $_z$

<i>condiciones</i>
$_y > _z$

QBE permite la aparición de expresiones aritméticas complejas dentro del cuadro de condiciones. Se puede formular la consulta “Determinar todas las sucursales que tienen activos que, como mínimo, dupliquen los activos de una de las sucursales con sede en Arganzuela” de manera parecida a como se ha formulado la consulta anterior y modificando el cuadro de condiciones:

<i>condiciones</i>
$_y \geq 2 * _z$

Para obtener el número de cuenta de todas las cuentas que tienen un saldo entre 1.300 € y 2.000 €, pero que no sea exactamente igual a 1.500 € se escribirá:

cuenta	número_cuenta	nombre_sucursal	saldo
P.			x

<i>condiciones</i>	
$\neg x = (\geq 1300 \text{ and } \leq 2000 \text{ and } \neg 1500)$	

QBE usa el constructor **or** de manera poco habitual para permitir la realización de comparaciones con conjuntos de valores constantes. Para determinar todas las sucursales que se hallan en Arganzuela o en Usera, se escribirá:

sucursal	nombre_sucursal	ciudad_sucursal	activo
P.		$\neg x$	

<i>condiciones</i>	
$\neg x = (\text{Arganzuela or Usera})$	

5.3.5 La relación resultado

Las consultas que se han formulado hasta ahora tienen una característica en común: los resultados aparecen en un único esquema de relación. Si el resultado de una consulta contiene atributos de varios esquemas de relación, se necesita un mecanismo para mostrar el resultado deseado en una sola tabla. Para este propósito se puede declarar una relación temporal *resultado* que incluya todos los atributos del resultado de la consulta. Se imprime el resultado deseado incluyendo el comando P. únicamente en el esqueleto de la tabla *resultado*.

Como ejemplo, considérese la consulta “Determinar *nombre_cliente*, *número_cuenta* y *saldo* de todas las cuentas de la sucursal de Navacerrada”. En el álgebra relacional esta consulta se formularía de la forma siguiente:

1. Reunión de las relaciones *impositor* y *cuenta*.
2. Proyección sobre los atributos *nombre_cliente*, *número_cuenta* y *saldo*.

Para formular la misma consulta en QBE se procede del siguiente modo:

1. Se crea un esqueleto de tablas, denominado *resultado*, con los atributos *nombre_cliente*, *número_cuenta* y *saldo*. El nombre del esqueleto de tabla recién creado (es decir, *resultado*), debe ser diferente de todos los nombres de relación de la base de datos ya existentes.
2. Se escribe la consulta.

La consulta resultante es:

cuenta	número_cuenta	nombre_sucursal	saldo
	$\neg y$	Navacerrada	$\neg z$

impositor	nombre_cliente	número_cuenta
	$\neg x$	$\neg y$

resultado	nombre_cliente	número_cuenta	saldo
P.	$\neg x$	$\neg y$	$\neg z$

5.3.6 QBE en Microsoft Access

En este apartado se revisa la versión de QBE soportada por Microsoft Access. Aunque QBE originalmente se diseñó para un entorno de visualización basado en texto, QBE de Access está diseñado para un

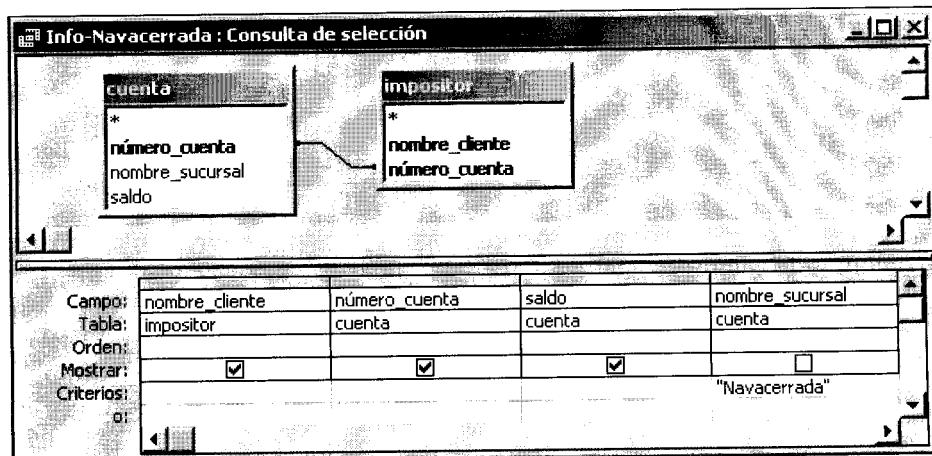


Figura 5.3 Una consulta de ejemplo en QBE de Microsoft Access.

entorno gráfico de visualización y, por tanto, se denomina **consulta gráfica mediante ejemplos (GQBE, Graphical Query-By-Example)**.

La Figura 5.3 muestra una consulta de ejemplo en GQBE. La consulta se puede describir en español como “Determinar *nombre_cliente*, *número_cuenta* y *saldo* de todas las cuentas de la sucursal de Navacerrada”. En el Apartado 5.3.5 se mostró cómo se expresa en QBE.

Una pequeña diferencia en la versión de GQBE es que los atributos de las tablas se escriben uno debajo de otro, en lugar de horizontalmente. Una diferencia más significativa es que la versión gráfica de QBE emplea una línea que une los atributos de dos tablas, en lugar de una variable compartida, para especificar las condiciones de reunión.

Una característica interesante de QBE de Access es que los vínculos entre las tablas se crean automáticamente según el nombre de los atributos. En el ejemplo de la Figura 5.3 se han integrado en la consulta las tablas *cuenta* e *impositor*. El atributo *número_cuenta* se comparte entre las dos tablas seleccionadas y el sistema inserta automáticamente un vínculo entre esas dos tablas. En otras palabras, de manera predeterminada se impone una condición de reunión natural entre las tablas; el vínculo se puede borrar si no se desea que exista. El vínculo también se puede especificar para que denote una reunión externa natural, en lugar de una reunión natural.

Otra pequeña diferencia en QBE de Access es que especifica en un cuadro separado, denominado **cuadrícula de diseño**, los atributos que se van a mostrar en lugar de usar P. en la tabla. En esta cuadrícula de diseño también se especifican las selecciones según el valor de los atributos.

Las consultas que implican agrupaciones y agregaciones se pueden crear en Access como se muestra en la Figura 5.4. La consulta de la figura busca el nombre, la dirección y la ciudad de todos los clientes que tienen más de una cuenta en el banco. Los atributos y las funciones “de agregación” se marcan en la cuadrícula de diseño.

Téngase en cuenta que cuando aparece alguna condición en una columna de la cuadrícula de diseño con la fila “Total” definida para una función de agregación, la condición se aplica sobre el valor agregado; por ejemplo, en la Figura 5.4, la selección “> 1” de la columna *número_cuenta* se aplica al resultado de la función de agregación “Contar”. Estas selecciones equivalen a las de las cláusulas **having** de SQL.

Las condiciones de selección se pueden aplicar a columnas de la cuadrícula de diseño que no estén ni agrupadas ni agregadas; estos atributos deben marcarse como “Dónde” en la fila “Total”. Este tipo de selecciones “Dónde” se aplican antes de la agregación, y equivalen a las selecciones en las cláusulas **where** de SQL. No obstante, este tipo de columnas no puede mostrarse (no es posible marcarlas como “Mostrar”). Sólo se pueden mostrar las columnas en que la fila “Total” especifica una “agrupación” o una función de agregación.

Las consultas se crean mediante una interfaz gráfica de usuario, seleccionando en primer lugar las tablas. A continuación se pueden añadir los atributos a la cuadrícula de diseño arrastrándolos y soltándolos desde las tablas. Las condiciones de selección, agrupación y agregación se pueden especificar a continuación sobre los atributos de la cuadrícula de diseño. QBE de Access ofrece otra serie de caracte-

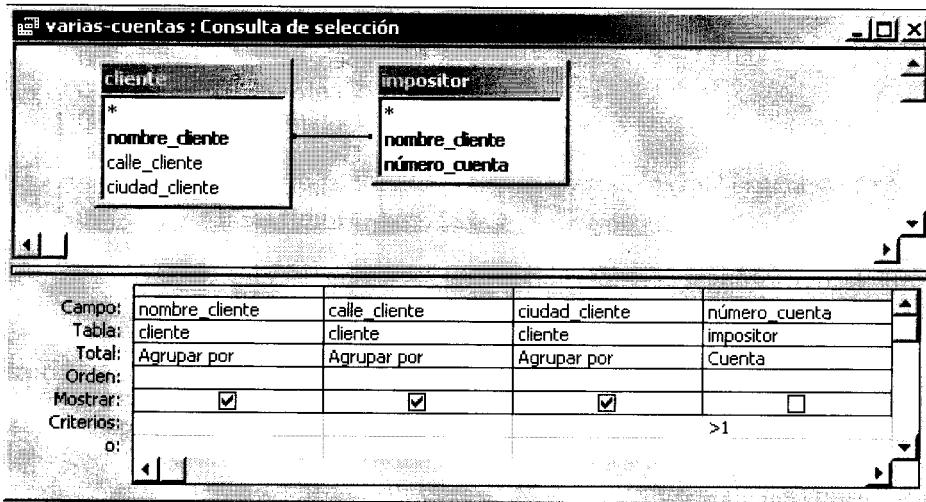


Figura 5.4 Una consulta de agregación en QBE de Microsoft Access.

rísticas, incluidas las consultas para la modificación de la base de datos mediante inserción, borrado y actualización.

5.4 Datalog

Datalog es un lenguaje de consultas no procedimental basado en el lenguaje de programación lógica Prolog. Al igual que en el cálculo relacional, el usuario describe la información deseada sin especificar el procedimiento concreto para obtener esa información. La sintaxis de Datalog recuerda la de Prolog. Sin embargo, el significado de los programas en Datalog se define de una manera puramente declarativa, a diferencia de la semántica más procedural de Prolog, por lo que Datalog simplifica la escritura de consultas sencillas y facilita la optimización de las consultas.

5.4.1 Estructura básica

Cada programa Datalog consiste en un conjunto de **reglas**. Antes de dar una definición formal de las reglas Datalog y de su significado formal, se presentarán algunos ejemplos. Supóngase una regla Datalog para definir una vista *v1* que contiene el número de cuenta y el saldo de las cuentas de la sucursal de Navacerrada cuyo saldo es superior a 700 €:

$$v1(C, S) :- cuenta(C, \text{"Navacerrada"}, S), S > 700$$

Las reglas Datalog definen vistas; la regla anterior **usa** la relación *cuenta* y **define** la vista *v1*. El símbolo **:-** se lee “si”, y la coma que separa “*cuenta(C, “Navacerrada”, S)*” de “*S > 700*” se lee “y”. Intuitivamente, la regla se entiende del siguiente modo:

```

for all C, S
if      (C, "Navacerrada", S) in cuenta and S > 700
then   (C, S) in v1
  
```

Supóngase que la relación *cuenta* es la mostrada en la Figura 5.5. Entonces, la vista *v1* contiene las tuplas mostradas en la Figura 5.6. Para obtener el saldo de la cuenta C-217 de esta vista se escribe:

$$? v1("C-217", S)$$

La respuesta a la consulta es:

$$(C-217, 750)$$

número_cuenta	nombre_sucursal	saldo
C-215	Becerril	700
C-101	Centro	500
C-305	Collado Mediano	350
C-201	Galapagar	900
C-217	Galapagar	750
C-222	Moralzarzal	700
C-102	Navacerrada	400

Figura 5.5 La relación *cuenta*.

Para determinar el número de cuenta y el saldo de todas las cuentas de la relación *v1* cuyo saldo es superior a 800 se puede escribir:

? *v1(C, S)*, *S* > 800

La respuesta a la consulta es:

(C-201, 900)

En general hace falta más de una regla para definir una vista. Cada regla define un conjunto de tuplas que debe contener la vista. El conjunto de tuplas de la vista se define, por tanto, como la unión de todos esos conjuntos de tuplas. El siguiente programa Datalog especifica los tipos de interés para las cuentas:

```
tipo_interés(C, 5) :- cuenta(C, N, S), S < 10000
tipo_interés(C, 6) :- cuenta(C, N, S), S >= 10000
```

El programa tiene dos reglas que definen la vista *tipo_interés*, cuyos atributos son el número de cuenta y el tipo de interés. Las reglas especifican que, si el saldo es menor de 10.000 €, el tipo de interés es el cinco por ciento y, si el saldo es igual o superior a 10.000 €, entonces el tipo de interés es el seis por ciento.

Las reglas Datalog pueden usar la negación. Las reglas siguientes definen una vista *c* que contiene el nombre de todos los clientes que tienen abierta una cuenta pero no tienen concedido ningún préstamo en el banco:

```
c(N) :- impositor(N,C), not es_prestatario(N)
es_prestatario(N) :- prestatario(N, P)
```

Prolog y la mayoría de las implementaciones de Datalog reconocen los atributos de una relación por su posición y omiten el nombre de los atributos. Por tanto, las reglas Datalog son compactas en comparación con las consultas de SQL. Sin embargo, cuando las relaciones tienen gran número de atributos o el orden o el número de los atributos de la relación pueden variar, la notación posicional puede resultar engorrosa y conducir a errores. No es difícil crear una variante de la sintaxis de Datalog que reconozca los atributos por su nombre en lugar de por su posición. En un sistema de ese tipo, la regla Datalog que define *v1*, se puede escribir como:

```
v1(número_cuenta C, saldo S) :-
    cuenta(número_cuenta C, nombre_sucursal "Navacerrada", saldo S),
    S > 700
```

número_cuenta	saldo
C-201	900
C-217	750

Figura 5.6 La relación *v1*.

```

interés(C, I) :- cuenta(C, "Navacerrada", S),
    tipo_interés(C, T), I = S * T / 100
tipo_interés(C, 5) :- cuenta(C, N, S), S < 10000
tipo_interés(C, 6) :- cuenta(C, N, S), S >= 10000

```

Figura 5.7 Programa Datalog que define el interés de las cuentas de la sucursal de Navacerrada.

La traducción entre las dos variedades puede hacerse sin un esfuerzo significativo, siempre que se disponga del esquema de relación.

5.4.2 Sintaxis de las reglas Datalog

Una vez que se han explicado informalmente las reglas y las consultas, se puede definir formalmente su sintaxis; su significado se estudia en el Apartado 5.4.3. Se usan los mismos convenios que en el álgebra relacional para denotar los nombres de las relaciones, de los atributos y de las constantes (números o cadenas de caracteres entrecomilladas). Se emplean letras mayúsculas y palabras con la primera letra en mayúsculas para denotar nombres de variables, y letras minúsculas y palabras con la primera letra en minúsculas para denotar los nombres de las relaciones y de los atributos. Algunos ejemplos de constantes son 4, que es un número, y "Martín", que es una cadena de caracteres; X y $Nombre$ son variables. Un **literal positivo** tiene la forma:

$$p(t_1, t_2, \dots, t_n)$$

donde p es el nombre de una relación con n atributos y t_1, t_2, \dots, t_n son constantes o variables. Un **literal negativo** tiene la siguiente forma:

$$\text{not } p(t_1, t_2, \dots, t_n)$$

donde la relación p tiene n atributos. El siguiente es un ejemplo de literal:

$$\text{cuenta}(C, "Navacerrada", S)$$

Los literales que contienen operaciones aritméticas se tratan de un modo especial. Por ejemplo, el literal $S > 700$, aunque no tiene la sintaxis descrita, puede entenderse conceptualmente como $>(S, 700)$, que sí tiene la sintaxis requerida y donde $>$ es una relación.

Pero ¿qué significa esta notación para las operaciones aritméticas como " $>$ "? La relación $>$ (conceptualmente) contiene tuplas de la forma (x, y) para todos los pares de valores x, y posibles, tales que $x > y$. Por tanto, $(2, 1)$ y $(5, -33)$ son tuplas de la relación $>$. Evidentemente, la relación $>$ es infinita. Otras operaciones aritméticas (como $>$, $=$, $+$ o $-$) se tratan también conceptualmente como relaciones. Por ejemplo, $A = B + C$ se puede tratar conceptualmente como $+(B, C, A)$, donde la relación $+$ contiene todas las tuplas (x, y, z) tales que $z = x + y$.

Un **hecho** se escribe de la forma:

$$p(v_1, v_2, \dots, v_n)$$

y denota que la tupla (v_1, v_2, \dots, v_n) pertenece a la relación p . Un conjunto de hechos de una relación se puede escribir también con la notación tabular habitual. Un conjunto de hechos para las relaciones de un esquema de base de datos equivale a un ejemplar del esquema de la base de datos. Las **reglas** se construyen a partir de literales, y tienen la forma:

$$p(t_1, t_2, \dots, t_n) :- L_1, L_2, \dots, L_n$$

donde cada L_i es un literal (positivo o negativo). El literal $p(t_1, t_2, \dots, t_n)$ se denomina **cabeza** de la regla y el resto de los literales constituyen el **cuerpo** de la misma.

Un **programa Datalog** consiste en un conjunto de reglas y el orden en el que se formulen es indiferente. Como ya se ha mencionado, puede haber varias reglas que definan cada relación.

En la Figura 5.7 se muestra un ejemplo de programa Datalog que define el tipo de interés para cada cuenta de la sucursal de Navacerrada. La primera regla del programa define la vista *interés*, cuyos atributos son el número de cuenta y su tipo de interés. Usa la relación *cuenta* y la vista *tipo_interés*. Las dos últimas reglas del programa son reglas que ya se han visto.

Se dice que la vista v_1 **depende directamente de** la vista v_2 si v_2 se usa en la expresión que define a v_1 . En el programa anterior, la vista *interés* depende directamente de las relaciones *tipo_interés* y *cuenta*. A su vez, la relación *tipo_interés* depende directamente de *cuenta*.

Se dice que la vista v_1 **depende indirectamente de** la vista v_2 si hay una secuencia de relaciones intermedias i_1, i_2, \dots, i_n para algún n tal que v_1 depende directamente de i_1 , i_1 depende directamente de i_2 , y así sucesivamente hasta i_{n-1} , que depende directamente de i_n .

En el ejemplo de la Figura 5.7, dado que hay una cadena de dependencias desde *interés*, pasando por *tipo_interés*, hasta *cuenta*, la relación *interés* también depende indirectamente de *cuenta*.

Finalmente, se dice que la vista v_1 **depende de** la vista v_2 si v_1 depende directa o indirectamente de v_2 .

Se dice que la vista v es **recursiva** si depende de sí misma. Se dice que la vista que no depende de sí misma **no es recursiva**.

Considérese el programa de la Figura 5.8. En él, la vista *empl* depende de sí misma (debido a la segunda regla) y, por tanto, es recursiva. En cambio, el programa de la Figura 5.7 no es recursivo.

5.4.3 Semántica de Datalog no recursivo

A continuación se considera la semántica formal de los programas Datalog. Por ahora se analizarán únicamente los programas no recursivos. La semántica de los programas recursivos es algo más complicada y se analizará en el Apartado 5.4.6. La semántica de los programas se define empezando por la semántica de una sola regla.

5.4.3.1 Semántica de las reglas

El **ejemplar básico de una regla** es el resultado de sustituir cada variable de la regla por alguna constante. Si una variable aparece varias veces en una regla, todas las apariciones de esa variable se deben sustituir por la misma constante. Los ejemplares básicos se suelen llamar simplemente **ejemplares**.

La regla de ejemplo que define $v1$ y un ejemplar de esa regla quedan como:

$$\begin{aligned} v1(A, B) &:= cuenta(C, "Navacerrada", S), S > 700 \\ v1("C-217", 750) &:= cuenta("C-217", "Navacerrada", 750), 750 > 700 \end{aligned}$$

En este ejemplo la variable C ha sido sustituida por "C-217" y la variable S por 750.

Normalmente cada regla tiene muchos ejemplares posibles. Estos ejemplares se corresponden con las diversas formas de asignar valores a cada variable de la regla.

Supóngase que se tiene la regla R :

$$p(t_1, t_2, \dots, t_n) := L_1, L_2, \dots, L_n$$

y un conjunto de hechos I asociados a las relaciones que aparecen en la regla (I también se puede considerar un ejemplar de la base de datos). Considérese cualquier ejemplar R' de la regla R :

$$p(v_1, v_2, \dots, v_n) := l_1, l_2, \dots, l_n$$

donde cada literal l_i es de la forma $q_i(v_{i,1}, v_{i,2}, \dots, v_{i,n_i})$ o **not** $q_i(v_{i,1}, v_{i,2}, \dots, v_{i,n_i})$, donde cada v_i y cada $v_{i,j}$ son constantes.

$$\begin{aligned} empl(X, Y) &:= jefe(X, Y) \\ empl(X, Y) &:= jefe(X, Z), empl(Z, Y) \end{aligned}$$

Figura 5.8 Programa Datalog recursivo.

Se dice que el cuerpo del ejemplar R' de la regla se **satisface** en I si:

1. Para cada literal positivo $q_i(v_{i,1}, \dots, v_{i,n_i})$ del cuerpo de R' , el conjunto de hechos I contiene el hecho $q(v_{i,1}, \dots, v_{i,n_i})$.
2. Para cada literal negativo **not** $q_j(v_{j,1}, \dots, v_{j,n_j})$ del cuerpo de R' , el conjunto de hechos I no contiene el hecho $q_j(v_{j,1}, \dots, v_{j,n_j})$.

Se define el conjunto de hechos que se pueden **inferir** a partir de un conjunto de hechos I dado empleando la regla R como:

$$\text{inferir}(R, I) = \{p(t_1, \dots, t_{n_i}) \mid \text{existe un ejemplar } R' \text{ de } R, \\ \text{donde } p(t_1, \dots, t_{n_i}) \text{ es la cabeza de } R', \text{ y} \\ \text{el cuerpo de } R' \text{ se satisface en } I\}.$$

Dado un conjunto de reglas $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$ se define:

$$\text{inferir}(\mathcal{R}, I) = \text{inferir}(R_1, I) \cup \text{inferir}(R_2, I) \cup \dots \cup \text{inferir}(R_n, I)$$

Supóngase un conjunto de hechos I que contiene las tuplas de la relación *cuenta* de la Figura 5.5. Un posible ejemplar de la regla de ejemplo R es

$$v1("C-217", 750) :- cuenta("C-217", "Navacerrada", 750), 750 > 700$$

El hecho *cuenta* ("C-217", "Navacerrada", 750) pertenece al conjunto de hechos I . Además, 750 es mayor que 700 y, por tanto, conceptualmente, (750, 700) pertenece a la relación " $>$ ". Por tanto, el cuerpo del ejemplar de la regla se satisface en I . Existen otros ejemplares posibles de R y, usándolos, se comprueba que $\text{inferir}(R, I)$ tiene exactamente el conjunto de hechos para $v1$ que se muestra en la Figura 5.9.

5.4.3.2 Semántica de los programas

Cuando una vista se define en términos de otra, el conjunto de hechos de la primera vista depende de los hechos de la segunda. En este apartado se da por supuesto que las definiciones no son recursivas: es decir, ninguna vista depende (directa o indirectamente) de sí misma. Por tanto, se pueden superponer las vistas en capas de la forma siguiente, y se puede emplear la superposición para definir la semántica del programa:

- Una relación está en la capa 1 si todas las relaciones que aparecen en los cuerpos de las reglas que la definen están almacenadas en la base de datos.
- Una relación está en la capa 2 si todas las relaciones que aparecen en los cuerpos de las reglas que la definen están almacenadas en la base de datos, o son de la capa 1.
- En general, una relación p está en la capa $i + 1$ si (1) no está en las capas $1, 2, \dots, i$ y (2) todas las relaciones que aparecen en los cuerpos de las reglas que la definen están almacenadas en la base de datos o son de las capas $1, 2, \dots, i$.

Considérese el programa de la Figura 5.7 con la regla adicional:

$$\text{cuenta_navacerrada}(X, Y) :- \text{cuenta}(X, "Navacerrada", Y)$$

<i>número_cuenta</i>	<i>saldo</i>
C-201	900
C-217	750

Figura 5.9 Resultado de $\text{inferir}(R, I)$.

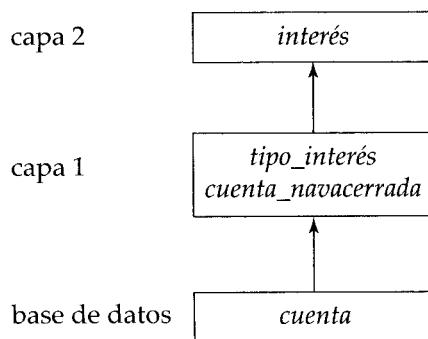


Figura 5.10 Clasificación en capas de las vistas.

La clasificación en capas de las vistas del programa se muestra en la Figura 5.10. La relación *cuenta* está en la base de datos. La relación *tipo_interés* está en la capa 1, mientras que todas las relaciones que se usan en las dos reglas que la definen están en la base de datos. La relación *cuenta_navacerrada* está también en la capa 1. Por último, la relación *interés* está en la capa 2, puesto que no está en la capa 1 y todas las relaciones que se usan en las reglas que la definen están en la base de datos o en capas inferiores a la 2.

Ahora se puede definir la semántica de los programas Datalog en términos de la clasificación en capas de las vistas. Sean las capas de un programa dado $1, 2, \dots, n$ y \mathcal{R}_i el conjunto de todas las reglas que definen vistas en la capa i .

- Se define I_0 como el conjunto de hechos almacenados en la base de datos e I_1 como

$$I_1 = I_0 \cup \text{inferir}(\mathcal{R}_1, I_0)$$

- Se procede de un modo análogo, definiendo I_2 en términos de I_1 y \mathcal{R}_2 , y así sucesivamente usando la siguiente definición:

$$I_{i+1} = I_i \cup \text{inferir}(\mathcal{R}_{i+1}, I_i)$$

- Por último, el conjunto de hechos de las vistas definidos por el programa (también denominado **semántica del programa**) se define como el conjunto de hechos I_n de la capa más alta n .

Para el programa de la Figura 5.7, I_0 es el conjunto de hechos en la base de datos e I_1 es el conjunto de hechos de la base de datos ampliado con el conjunto de todos los hechos que se pueden inferir de I_0 usando las reglas de las relaciones *tipo_interés* y *cuenta_navacerrada*. Finalmente, I_2 contiene los hechos de I_1 junto con los hechos de la relación *interés* que se pueden inferir de los hechos en I_1 mediante la regla que define *interés*. La semántica del programa—es decir, el conjunto de hechos que están en cada vista—se define como el conjunto de hechos de I_2 .

Recuérdese que en el Apartado 3.9.2 se vio la manera de definir el significado de las vistas no recursivas del álgebra relacional empleando una técnica denominada **expansión de vistas**. La expansión de vistas se puede usar también con las vistas no recursivas de Datalog; del mismo modo, la técnica de clasificación por capas aquí descrita se puede usar con las vistas del álgebra relacional.

5.4.4 Seguridad

Es posible formular reglas que generen un número infinito de respuestas. Considérese la regla

$$\text{mayor}(X, Y) :- X > Y$$

Como la relación que define $>$ es infinita, esta regla generaría un número infinito de hechos para la relación *mayor*, cuyo cálculo necesitaría, lógicamente, una cantidad infinita de tiempo y de espacio.

El empleo de la negación puede causar problemas parecidos. Considérese la regla:

$$\text{no_en_préstamo}(P, S, I) :- \text{not } \text{préstamo}(P, S, I)$$

La idea es que la tupla $(número_préstamo, nombre_sucursal, importe)$ está en la vista $no_en_préstamo$ si no pertenece a la relación $préstamo$. Sin embargo, si el conjunto de posibles números de préstamos, nombres de sucursales e importes es infinito, la relación $no_en_préstamo$ también será infinita.

Por último, si existe una variable en la cabeza de la regla que no aparece en el cuerpo, se puede generar un conjunto infinito de hechos en los que la variable se asigna a distintos valores.

Con objeto de evitar estas posibilidades, se exige que las reglas Datalog cumplan las siguientes condiciones de **seguridad**:

1. Todas las variables que aparecen en la cabeza de una regla deben aparecer en un literal positivo no aritmético en el cuerpo de esa regla.
2. Todas las variables que aparecen en un literal negativo en el cuerpo de una regla deben aparecer también en algún literal positivo en el cuerpo de la misma.

Si todas las reglas de un programa Datalog no recursivo satisfacen las condiciones de seguridad anteriores, se puede probar que todas las vistas definidas en el programa son finitas, siempre que todas las relaciones de la base de datos sean finitas. Estas condiciones se pueden relajar algo para permitir que las variables de la cabeza aparezcan sólo en literales aritméticos del cuerpo en determinados casos. Por ejemplo, en la regla:

$$p(A) :- q(B), A = B + 1$$

se puede observar que si la relación q es finita, también lo es p , por las propiedades de la suma, aunque la variable A sólo aparezca en un literal aritmético.

5.4.5 Operaciones relacionales en Datalog

Las expresiones de Datalog no recursivas sin operaciones aritméticas son equivalentes en poder expresivo a las expresiones que usan las operaciones básicas del álgebra relacional (\cup , $-$, \times , σ , Π y ρ). Esta afirmación no se probará formalmente ahora. En su lugar, se usarán ejemplos para mostrar la forma de expresar en Datalog las diversas operaciones del álgebra relacional. En todos los casos se define una vista denominada *consulta* para ilustrar las operaciones.

Ya se ha visto anteriormente la manera de llevar a cabo selecciones mediante reglas Datalog. Las proyecciones se realizan usando únicamente los atributos requeridos en la cabeza de la regla. Para proyectar el atributo *nombre_cuenta* de la relación *cuenta*, se usa:

$$\text{consulta}(C) :- \text{cuenta}(C, N, S)$$

Se puede obtener el producto cartesiano de dos relaciones r_1 y r_2 en Datalog de la manera siguiente:

$$\text{consulta}(X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m) :- r_1(X_1, X_2, \dots, X_n), r_2(Y_1, Y_2, \dots, Y_m)$$

donde r_1 es de aridad n , r_2 es de aridad m y $X_1, X_2, \dots, X_n, Y_1, Y_2, \dots, Y_m$ son nombres de variables, todos distintos entre sí.

La unión de dos relaciones r_1 y r_2 (ambas de aridad n), se forma del siguiente modo:

$$\begin{aligned} \text{consulta}(X_1, X_2, \dots, X_n) &:- r_1(X_1, X_2, \dots, X_n) \\ \text{consulta}(X_1, X_2, \dots, X_n) &:- r_2(X_1, X_2, \dots, X_n) \end{aligned}$$

El conjunto diferencia de dos relaciones r_1 y r_2 , se calcula como sigue:

$$\text{consulta}(X_1, X_2, \dots, X_n) :- r_1(X_1, X_2, \dots, X_n), \text{not } r_2(X_1, X_2, \dots, X_n)$$

Finalmente, obsérvese que, con la notación posicional usada en Datalog, el operador renombramiento ρ no es necesario. Cada relación puede aparecer más de una vez en el cuerpo de la regla pero, en lugar de renombrarla para dar nombres diferentes a sus apariciones, se emplean diferentes nombres de variables en las diferentes apariciones.

Es posible demostrar que cualquier consulta no recursiva de Datalog sin funciones aritméticas se puede expresar mediante las operaciones del álgebra relacional. Esta demostración se plantea como ejercicio para el lector. Por tanto, se puede establecer la equivalencia de las operaciones básicas del álgebra relacional y de Datalog no recursivo sin operaciones aritméticas.

Algunas extensiones de Datalog soportan las operaciones relacionales de actualización (inserción, borrado y actualización). La sintaxis para estas operaciones varía de una implementación a otra. Algunos sistemas permiten el uso de + o – en la cabeza de las reglas para denotar la inserción y el borrado relacionales. Por ejemplo, se pueden trasladar todas las cuentas de la sucursal de Navacerrada a la sucursal de Soto del Real ejecutando:

```
+ cuenta(C, "Soto del Real", S) :- cuenta(C, "Navacerrada", S)
- cuenta(C, "Navacerrada", S) :- cuenta(C, "Navacerrada", S)
```

Algunas implementaciones de Datalog también soportan la operación de agregación del álgebra relacional extendido. Una vez más, no existe sintaxis normalizada para esta operación.

5.4.6 Recursividad en Datalog

Varias aplicaciones de bases de datos manejan estructuras parecidas a los árboles de datos. Por ejemplo, considérense los empleados de una empresa. Algunos empleados son jefes. Cada jefe dirige un conjunto de personas que están bajo su mando. Pero cada una de ellas puede ser asimismo jefe y tener otras personas a su mando. Por tanto, los empleados se pueden organizar en una estructura parecida a un árbol.

Supóngase que se tiene el esquema de relación:

$$\text{Esquema_jefe} = (\text{nombre_empleado}, \text{nombre_jefe})$$

y sea *jefe* una relación del esquema anterior.

Supóngase ahora que se desea determinar los empleados que supervisa (directa o indirectamente) un determinado jefe—por ejemplo, Santos. Es decir, si el jefe de Alández es Bariego, el jefe de Bariego es Erice y el jefe de Erice es Santos, entonces Alández, Bariego y Erice son los empleados supervisados por Santos. A menudo se escriben programas recursivos para trabajar con los árboles de datos. Usando la idea de la recursividad se puede definir el conjunto de empleados supervisados por Santos como se indica a continuación. Las personas supervisadas por Santos son (1) las personas cuyo jefe directo es Santos y (2) las personas cuyo jefe es supervisado por Santos. Obsérvese que el caso (2) es recursivo.

Se puede formular la definición recursiva anterior como una vista recursiva de Datalog, denominada *empl_santos*:

$$\begin{aligned} \text{empl_santos}(X) &:- \text{jefe}(X, "Santos") \\ \text{empl_santos}(X) &:- \text{jefe}(X, Y), \text{empl_santos}(Y) \end{aligned}$$

La primera regla corresponde al caso (1) y la segunda al caso (2). La vista *empl_santos* depende de sí misma debido a la segunda regla; por tanto, el programa Datalog anterior es recursivo. Se da *por supuesto* que los programas Datalog recursivos no contienen reglas con literales negativos. La razón se verá más adelante. Las notas bibliográficas hacen referencia a artículos que describen dónde se puede usar la negación en los programas Datalog recursivos.

```
procedure PuntoFijo-Datalog
  I = conjunto de hechos de la base de datos
  repeat
    I_Anterior = I
    I = I ∪ inferir(R, I)
  until I = I_Anterior
```

Figura 5.11 Procedimiento PuntoFijo-Datalog.

nombre_empleado	nombre_jefe
Alández	Bariego
Bariego	Erice
Corisco	Dalma
Dalma	Santos
Erice	Santos
Santos	Marchamalo
Rienda	Marchamalo

Figura 5.12 La relación *jefe*.

Número de iteración	Tuplas de <i>empl_santos</i>
0	
1	(Dalma), (Erice)
2	(Dalma), (Erice), (Bariego), (Corisco)
3	(Dalma), (Erice), (Bariego), (Corisco), (Alández)
4	(Dalma), (Erice), (Bariego), (Corisco), (Alández)

Figura 5.13 Subordinados de Santos en las distintas iteraciones del procedimiento PuntoFijo-Datalog.

Las vistas de los programas recursivos que contienen un conjunto de reglas \mathcal{R} se definen para que contengan exactamente el conjunto de hechos I calculados por el procedimiento iterativo PuntoFijo-Datalog de la Figura 5.11. La recursividad del programa Datalog se ha transformado en la iteración del procedimiento. Al final del procedimiento, $\text{inferir}(\mathcal{R}, I) \cup D = I$, donde D es el conjunto de hechos de la base de datos, e I se denomina **punto fijo** del programa.

Considérese el programa que define *empl_santos* con la relación *jefe*, como en la Figura 5.12. El conjunto de hechos calculado para la vista *empl_santos* en cada iteración aparece en la Figura 5.13. En cada iteración el programa calcula otro nivel de empleados bajo el mando de Santos y los añade al conjunto *empl_santos*. El procedimiento termina cuando no se produce ningún cambio en el conjunto *empl_santos*, lo cual detecta el sistema al descubrir que $I = I_{\text{Anterior}}$. Como el conjunto de empleados y jefes es finito, se tiene que alcanzar este punto fijo. En la relación *jefe* dada, el procedimiento PuntoFijo-Datalog termina después de la cuarta iteración, al detectar que no se ha inferido ningún hecho nuevo.

Conviene comprobar que, al final de la iteración, la vista *empl_santos* contiene exactamente los empleados que trabajan bajo la supervisión de Santos. Para obtener los nombres de los empleados supervisados por Santos definidos por la vista se puede usar la consulta:

? *empl_santos*(N)

Para comprender el procedimiento PuntoFijo-Datalog hay que recordar que cada regla infiere hechos nuevos a partir de un conjunto de hechos dado. La iteración comienza con un conjunto de hechos I que coincide con los hechos de la base de datos. Se sabe que todos estos hechos son ciertos, pero puede haber otros hechos que también sean ciertos¹. A continuación se usa el conjunto de reglas \mathcal{R} del programa Datalog para inferir los hechos que son ciertos, dado que los hechos de I lo son. Los hechos inferidos se añaden a I y se vuelven a usar las reglas para hacer más inferencias. Este proceso se repite hasta que no se puedan inferir hechos nuevos.

Se puede demostrar para los programas Datalog seguros que existe un punto a partir del cual no se pueden obtener más hechos nuevos; es decir, para algún k , $I_{k+1} = I_k$. En ese punto, por tanto, se tiene el conjunto final de hechos ciertos. Además, dado un programa Datalog y una base de datos, el procedimiento de punto fijo infiere todos los hechos que se puede inferir que son ciertos.

1. La palabra “hecho” se usa en un sentido técnico para denotar la pertenencia de una tupla a una relación. Así, en el sentido de Datalog para “hecho”, un hecho puede ser cierto (la tupla está realmente en la relación) o falso (la tupla no está en la relación).

Si un programa recursivo contiene una regla con un literal negativo, puede surgir el problema siguiente. Recuérdese que cuando se hace una inferencia empleando el ejemplar básico de una regla, por cada literal negativo `not q` en el cuerpo de la regla hay que comprobar que `q` no esté presente en el conjunto de hechos I . Esta comprobación da por supuesto que `q` no se puede inferir posteriormente. Sin embargo, en la iteración de punto fijo, el conjunto de hechos I crece en cada iteración y, aunque `q` no esté presente en I en una iteración dada, puede aparecer más tarde. Por tanto, se puede haber hecho una inferencia en una iteración que ya no se pueda hacer en una iteración anterior, y la inferencia sería incorrecta. Se exige que los programas recursivos no contengan literales negativos para evitar estos problemas.

En lugar de crear una vista para los empleados supervisados por un jefe concreto, por ejemplo Santos, se puede crear una vista más general, `empl`, que contenga todas las tuplas (X, Y) tales que X sea directa o indirectamente supervisado por Y , usando el siguiente programa (que también puede verse en la Figura 5.8):

$$\begin{aligned} \textit{empl}(X, Y) &:- \textit{jefe}(X, Y) \\ \textit{empl}(X, Y) &:- \textit{jefe}(X, Z), \textit{empl}(Z, Y) \end{aligned}$$

Para determinar los subordinados directos o indirectos de Santos, se usa simplemente la consulta:

$$? \textit{empl}(X, \text{"Santos"})$$

que devuelve para X el mismo conjunto de valores que la vista `empl_santos`. La mayoría de las implementaciones de Datalog cuenta con sofisticados optimizadores de consultas y motores de evaluación que pueden ejecutar la consulta anterior aproximadamente a la misma velocidad que evaluarían la vista `empl_santos`. La vista `empl` definida anteriormente se denomina **cierre transitivo** de la relación `jefe`. Si se sustituyera la relación `jefe` por cualquier otra relación binaria R , el programa anterior definiría el cierre transitivo de R .

5.4.7 La potencia de la recursividad

Datalog con recursividad tiene mayor potencia expresiva que Datalog sin recursividad. En otras palabras, existen consultas a la base de datos que se pueden resolver usando recursividad, pero que no se pueden resolver sin usarla. Por ejemplo, en Datalog no se puede expresar el cierre transitivo sin usar la recursividad (como, por cierto, en SQL o QBE sin recursividad). Considérese el cierre transitivo de la relación `jefe`. Intuitivamente, un número fijo de reuniones sólo puede determinar los empleados que están un número fijo (diferente) de niveles por debajo de cualquier jefe (no se intentará demostrar esta afirmación aquí). Como cualquier consulta no recursiva tiene un número fijo de reuniones, existe un límite al número de niveles de empleados que la consulta puede determinar. Si el número de niveles de empleados de la relación `jefe` es mayor que el límite de la consulta, la consulta no encontrará algún nivel de empleados. Por tanto, los programas Datalog no recursivos no pueden expresar el cierre transitivo.

Una alternativa a la recursividad es usar un mecanismo externo, como SQL incorporado, para iterar por las consultas no recursivas. La iteración implementa el bucle del algoritmo de punto fijo de la Figura 5.11. De hecho, así es como se implementa ese tipo de consultas en los sistemas de bases de datos que no permiten la recursividad. Sin embargo, la formulación de estas consultas empleando la iteración es más complicada que si se usa la recursividad, y la evaluación usando recursividad puede optimizarse para que se ejecute más rápidamente que la evaluación mediante iteración.

La potencia expresiva proporcionada por la recursividad debe usarse con cuidado. Resulta relativamente sencillo escribir programas recursivos que generen un número infinito de hechos, como se muestra en este programa:

$$\begin{aligned} \textit{número}(0) \\ \textit{número}(A) &:- \textit{número}(B), A = B + 1 \end{aligned}$$

El programa genera `número(n)` para todos los enteros n positivos, que es claramente un conjunto infinito y no termina nunca. La segunda regla del programa no cumple la condición de seguridad descrita en

el Apartado 5.4.4. Los programas que cumplen la condición de seguridad terminan, aunque sean recursivos, siempre que todas las relaciones de la base de datos sean finitas. Para programas de este tipo, las tuplas de las vistas sólo pueden contener constantes de la base de datos y, por ello, las vistas son finitas. Lo opuesto no es cierto; es decir, hay programas que no cumplen la condición de seguridad, pero que finalizan.

El procedimiento PuntoFijo-Datalog usa de manera iterativa la función $\text{inferir}(\mathcal{R}, I)$ para calcular los hechos que son ciertos dado un programa Datalog recursivo. Aunque sólo se ha considerado el caso de los programas Datalog sin literales negativos, el procedimiento se puede emplear también en vistas definidas en otros lenguajes, como SQL o el álgebra relacional, siempre que las vistas cumplan las condiciones que se describen a continuación. Independientemente del lenguaje empleado para definir una vista V , esa vista se puede considerar definida por una expresión E_V que, dado un conjunto de hechos I , devuelve un conjunto de hechos $E_V(I)$ para la vista V . Dado un conjunto de definiciones de vistas \mathcal{R} (en cualquier lenguaje) se puede definir la función $\text{inferir}(\mathcal{R}, I)$ que devuelva $I \cup \bigcup_{V \in \mathcal{R}} E_V(I)$. La función anterior es parecida a la función inferir de Datalog.

Se dice que una vista V es **monótona** si, dado cualquier par de conjuntos de hechos I_1 e I_2 tales que $I_1 \subseteq I_2$, entonces $E_V(I_1) \subseteq E_V(I_2)$, donde E_V es la expresión usada para definir V . Análogamente, se dice que la función inferir es monótona si:

$$I_1 \subseteq I_2 \Rightarrow \text{inferir}(\mathcal{R}, I_1) \subseteq \text{inferir}(\mathcal{R}, I_2)$$

Por tanto, si inferir es monótona, dado un conjunto de hechos I_0 que es un subconjunto de los hechos ciertos, se puede asegurar que todos los hechos de $\text{inferir}(\mathcal{R}, I_0)$ también son ciertos. Usando el mismo razonamiento del Apartado 5.4.6 se puede demostrar que el procedimiento PuntoFijo-Datalog es correcto (es decir, sólo calcula hechos ciertos) siempre que la función inferir sea monótona.

Las expresiones del álgebra relacional que sólo usan los operadores $\Pi, \sigma, \times, \bowtie, \cup, \cap$, o ρ son monótonas. Se pueden definir vistas recursivas usando estas expresiones.

Sin embargo, las expresiones relacionales que usan el operador $-$ no son monótonas. Por ejemplo, sean $jefe_1$ y $jefe_2$ relaciones con el mismo esquema que la relación $jefe$. Sea

$$I_1 = \{ jefe_1("Alández", "Bariego"), jefe_1("Bariego", "Erice"), \\ jefe_2("Alández", "Bariego") \}$$

y sea

$$I_2 = \{ jefe_1("Alández", "Bariego"), jefe_1("Bariego", "Erice"), \\ jefe_2("Alández", "Bariego"), jefe_2("Bariego", "Erice") \}$$

Considérese la expresión $jefe_1 - jefe_2$. El resultado de la expresión anterior sobre I_1 es $(\text{"Bariego"}, \text{"Erice"})$, mientras que el resultado de la expresión sobre I_2 es la relación vacía. Pero, como $I_1 \subseteq I_2$, la expresión no es *monótona*. Las expresiones que usan el operador de agrupación del álgebra relacional extendido tampoco son monótonas.

La técnica del punto fijo no funciona sobre las vistas recursivas definidas con expresiones no monótonas. No obstante, existen situaciones en las que ese tipo de vistas es útil, especialmente para la definición de agregaciones en las relaciones “objeto–componente”. Ese tipo de relaciones define los componentes que constituyen cada objeto. Los componentes pueden estar formados, a su vez, por muchos otros componentes, y así sucesivamente; por tanto, las relaciones, como la relación $jefe$, tienen una estructura recursiva natural. Un ejemplo de consulta de agregación sobre una estructura de ese tipo es el cálculo del número total de componentes de cada objeto. Escribir esta consulta en Datalog o en SQL (sin las extensiones procedimentales) exigiría el empleo de una vista recursiva sobre una expresión no monótona. Las notas bibliográficas contienen referencias sobre la investigación acerca de la definición de este tipo de vistas.

Es posible definir algunos tipos de consultas recursivas sin usar vistas. Por ejemplo, se han propuesto operaciones relacionales extendidas para definir el cierre transitivo, y extensiones de la sintaxis de SQL para especificar el cierre transitivo (generalizado). Sin embargo, las definiciones de vistas recursivas proporcionan una mayor potencia expresiva que las demás formas de consultas recursivas.

5.5 Resumen

- El cálculo relacional de tuplas y el cálculo relacional de dominios son lenguajes no procedimentales que representan la potencia básica necesaria en un lenguaje de consultas relacionales. El álgebra relacional básica es un lenguaje procedural que es equivalente en potencia a ambas formas del cálculo relacional cuando se limitan a expresiones seguras.
- Los cálculos relacionales son lenguajes rígidos y formales que no resultan adecuados para los usuarios ocasionales de los sistemas de bases de datos. Los sistemas comerciales de bases de datos, por tanto, usan lenguajes con más “azúcar sintáctico”. Se han considerado dos lenguajes de consultas: QBE y Datalog.
- QBE está basado en un paradigma visual: las consultas tienen un aspecto muy parecido a tablas.
- QBE y sus variantes se han hecho populares entre los usuarios poco expertos de bases de datos, debido a la simplicidad intuitiva del paradigma visual. El muy usado sistema de bases de datos Access de Microsoft soporta una versión gráfica de QBE denominada GQBE.
- Datalog procede de Prolog pero, a diferencia de éste, tiene una semántica declarativa que hace que las consultas sencillas sean más fáciles de formular y que la evaluación de las consultas resulte más fácil de optimizar.
- La definición de las vistas resulta especialmente sencilla en Datalog, y las vistas recursivas que permite Datalog hacen posible la formulación de consultas, tales como el cierre transitivo, que no podrían formularse sin usar la recursividad o la iteración. Sin embargo, en Datalog no hay normas aceptadas para características importantes como la agrupación y la agregación. Datalog sigue siendo, principalmente, un lenguaje de investigación.

Términos de repaso

- Cálculo relacional de tuplas.
- Cálculo relacional de dominios.
- Seguridad de las expresiones.
- Potencia expresiva de los lenguajes.
- Query-by-Example (QBE, consulta mediante ejemplos).
- Sintaxis bidimensional.
- Esqueletos de tablas.
- Filas de ejemplo.
- Cuadro de condiciones.
- Relación resultado.
- Microsoft Access.
- Graphical Query-By-Example (GQBE, consulta gráfica mediante ejemplos).
- Cuadrícula de diseño.
- Datalog.
- Reglas.
- Usa.
- Define.
- Literal positivo.
- Literal negativo.
- Hecho.
- Regla:
 - Cabeza.
 - Cuerpo.
- Programa Datalog.
- Depende:
 - Directamente.
 - Indirectamente.
- Vista recursiva.
- Vista no recursiva.
- Ejemplares:
 - Básicos.
 - Satisfacer.
- Inferir.
- Semántica:
 - De las reglas.
 - De los programas.
- Seguridad.
- Punto fijo.
- Cierre transitivo.
- Definición de vistas monótonas.

Ejercicios prácticos

5.1 Sean los siguientes esquemas de relaciones:

$$\begin{aligned} R &= (A, B, C) \\ S &= (D, E, F) \end{aligned}$$

Sean las relaciones $r(R)$ y $s(S)$. Dese una expresión del cálculo relacional de tuplas que sea equivalente a cada una de las siguientes:

- a. $\Pi_A(r)$
- b. $\sigma_{B=17}(r)$
- c. $r \times s$
- d. $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

5.2 Sea $R = (A, B, C)$ y sean r_1 y r_2 relaciones del esquema R . Dese una expresión del cálculo relacional de dominios que sea equivalente a cada una de las siguientes:

- a. $\Pi_A(r_1)$
- b. $\sigma_{B=17}(r_1)$
- c. $r_1 \cup r_2$
- d. $r_1 \cap r_2$
- e. $r_1 - r_2$
- f. $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

5.3 Sean $R = (A, B)$ y $S = (A, C)$ y sean $r(R)$ y $s(S)$ relaciones. Escríbanse expresiones de QBE y de Datalog para cada una de las consultas siguientes:

- a. $\{< a > \mid \exists b (< a, b > \in r \wedge b = 7)\}$
- b. $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$
- c. $\{< a > \mid \exists c (< a, c > \in s \wedge \exists b_1, b_2 (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

5.4 Considérese la base de datos relacional de la Figura 5.14, en la que se han subrayado las claves primarias. Proporcionese una expresión de Datalog para cada una de las siguientes consultas:

- a. Determinar todos los empleados que trabajan (directa o indirectamente) a las órdenes del jefe “Santos”.
- b. Determinar las ciudades de residencia de todos los empleados que trabajan (directa o indirectamente) a las órdenes del jefe “Santos”.
- c. Determinar todas las parejas de empleados que tienen un jefe (directo o indirecto) en común.
- d. Determinar todas las parejas de empleados que tienen un jefe (directo o indirecto) en común y que están el mismo número de niveles por debajo de ese jefe.

5.5 Describábase cómo una regla Datalog arbitraria puede expresarse como una vista del álgebra relacional extendida.

Ejercicios

5.6 Considérese la base de datos de empleados de la Figura 5.14. Proporcionense expresiones del cálculo relacional de tuplas para cada una de las consultas siguientes:

- a. Determinar el nombre de todos los empleados que trabajan en el Banco Importante.
- b. Determinar el nombre y la ciudad de residencia de todos los empleados que trabajan en el Banco Importante.
- c. Determinar el nombre, la dirección y la ciudad de residencia de todos los empleados que trabajan en el Banco Importante y ganan más de 10.000 € anuales.
- d. Determinar todos los empleados que viven en la ciudad en la que se ubica la empresa para la que trabajan.
- e. Determinar todos los empleados que viven en la misma ciudad y en la misma calle que su jefe.
- f. Determinar todos los empleados de la base de datos que no trabajan en el Banco Importante.
- g. Determinar todos los empleados que ganan más que cualquier empleado del Banco Pequeño.

*empleado (nombre_empleado, calle, ciudad)
trabaja (nombre_empleado, nombre_empresa, sueldo)
empresa (nombre_empresa, ciudad)
jefe (nombre_empleado, nombre_jefe)*

Figura 5.14 Base de datos de empleados.

- h. Supóngase que las empresas pueden tener sede en varias ciudades. Determinar todas las empresas con sede en todas las ciudades en las que tiene sede el Banco Pequeño.
- 5.7 Sea $R = (A, B)$ y $S = (A, C)$ y sean $r(R)$ y $s(S)$ relaciones. Escríbanse expresiones del álgebra relacional equivalentes a cada una de las expresiones siguientes del cálculo relacional de dominios:
- $\{< a > \mid \exists b (< a, b > \in r \wedge b = 17)\}$
 - $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$
 - $\{< a > \mid \exists b (< a, b > \in r) \vee \forall c (\exists d (< d, c > \in s) \Rightarrow < a, c > \in s)\}$
 - $\{< a > \mid \exists c (< a, c > \in s \wedge \exists b_1, b_2 (< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$
- 5.8 Repítase el Ejercicio 5.7 escribiendo consultas SQL en lugar de expresiones del álgebra relacional.
- 5.9 Sea $R = (A, B)$ y $S = (A, C)$ y sean $r(R)$ y $s(S)$ relaciones. Usando la constante especial *nulo*, escríbanse expresiones del cálculo relacional de tuplas equivalentes a cada una de las siguientes:
- $r \bowtie s$
 - $r \bowtie \bowtie s$
 - $r \bowtie \bowtie s$
- 5.10 Considérese la base de datos de seguros de la Figura 5.15, en la que las claves primarias se han subrayado. Formúlense las siguientes consultas en GQBE para esta base de datos relacional:
- Determinar el número total de personas que poseen coches que se hayan visto involucrados en accidentes en 2005.
 - Determinar el número de accidentes en los cuales se ha visto involucrado algún coche perteneciente a “Martín Gómez”.
- 5.11 Dese una expresión del cálculo relacional de tuplas para determinar el valor máximo de $r(A)$.
- 5.12 Repítase el ejercicio 5.6 empleando QBE y Datalog.
- 5.13 Sea $R = (A, B, C)$ y sean r_1 y r_2 relaciones del esquema R . Dense expresiones en QBE y Datalog equivalentes a cada una de las consultas siguientes:
- $r_1 \cup r_2$
 - $r_1 \cap r_2$
 - $r_1 - r_2$
 - $\Pi_{AB}(r_1) \bowtie \Pi_{BC}(r_2)$
- 5.14 Escríbase una vista del álgebra relacional extendida equivalente a la regla Datalog

$$p(A, C, D) := q1(A, B), q2(B, C), q3(4, B), D = B + 1$$

*persona (número_carné, nombre, dirección)
coche (matrícula, modelo, año)
accidente (número_parte, fecha, lugar)
es_dueño (número_carné, matrícula)
participó (número_carné, coche, número_parte, importe_daños)*

Figura 5.15 Base de datos de seguros.

Notas bibliográficas

La definición original del cálculo relacional de tuplas aparece en Codd [1972]. Hay una prueba formal de la equivalencia del cálculo relacional de tuplas y del álgebra relacional en Codd [1972]. Se han propuesto varias extensiones del cálculo relacional de tuplas. Klug [1982] y Escobar-Molano et al. [1993] describen extensiones para funciones de agregación escalares.

El sistema de bases de datos QBE se desarrolló en el Centro de Investigación T. J. Watson de IBM en los primeros años 1970. El lenguaje de manipulación de datos de QBE se usó más tarde en el Servicio de Gestión de Consultas (Query Management Facility, QMF) de IBM. La versión original de Query-by-Example se describe en Zloof [1977]. Otras implementaciones de QBE son Access de Microsoft y Paradox de Borland (que ya no tiene soporte técnico).

Ullman [1988] y Ullman [1989] ofrecen amplias explicaciones teóricas sobre los lenguajes lógicos de consultas y sus técnicas de implementación. Ramakrishnan y Ullman [1995] ofrecen un resumen más actual de las bases de datos deductivas.

A los programas Datalog que tienen tanto recursividad como negación se les puede asignar una semántica sencilla si las negaciones están “estratificadas”—es decir, si no hay ninguna recursividad que afecte a las negaciones. Chandra y Harel [1982] y Apt y Pugin [1987] estudian las negaciones estratificadas. Una extensión importante, denominada *semántica de la estratificación modular*, que maneja una clase de programas recursivos con literales negativos, se estudia en Ross [1990]; Ramakrishnan et al. [1992] describe una técnica de evaluación para este tipo de programas.

Herramientas

QBE de Access de Microsoft es actualmente la implementación de QBE más usada. Las ediciones QMF y Everywhere de DB2 de IBM también soportan QBE.

El sistema Coral de la Universidad de Wisconsin–Madison (www.cs.wisc.edu/coral) es una implementación de Datalog. El sistema XSB de la Universidad Estatal de Nueva York (SUNY, State University of New York) en Stony Brook (xsb.sourceforge.net) es una implementación de Prolog muy usada que soporta consultas a las bases de datos; recuérdese que Datalog es un subconjunto no procedimental de Prolog.

Diseño de bases de datos

Los sistemas de bases de datos están diseñados para gestionar grandes cantidades de información. Estas grandes cantidades de información no se encuentran aisladas. Forman parte del funcionamiento de alguna empresa cuyo producto final puede ser la información obtenida de la base de datos o algún producto o servicio para el que la base de datos sólo desempeña un papel secundario.

Los dos primeros capítulos de esta parte se centran en el diseño de los esquemas de las bases de datos. El modelo entidad-relación (E-R) descrito en el Capítulo 6 es un modelo de datos de alto nivel. En lugar de representar todos los datos en tablas, distingue entre los objetos básicos, denominados *entidades*, y las *relaciones* entre esos objetos. Suele utilizarse como un primer paso en el diseño de los esquemas de las bases de datos.

El diseño de las bases de datos relacionales—el diseño del esquema relacional—se trató informalmente en los capítulos anteriores. No obstante, existen criterios para distinguir los buenos diseños de bases de datos de los malos. Éstos se formalizan mediante varias “formas normales” que ofrecen diferentes compromisos entre la posibilidad de inconsistencias y la eficiencia de determinadas consultas. El Capítulo 7 describe el diseño formal de los esquemas de las relaciones.

El diseño de un entorno completo de aplicaciones para bases de datos, que responda a las necesidades de la empresa que se modela, exige prestar atención a un conjunto de aspectos más amplio, muchos de los cuales se tratan en el Capítulo 8. Este capítulo describe las interfaces para bases de datos basadas en Web y amplía el estudio de capítulos anteriores sobre la integridad de los datos y la seguridad.

Diseño de bases de datos y el modelo E-R

Hasta este punto se ha dado por supuesto un determinado esquema de la base de datos y se ha estudiado la manera de expresar las consultas y las actualizaciones. Ahora se va a considerar en primer lugar la manera de diseñar el esquema de la base de datos. En este capítulo nos centraremos en el modelo de datos entidad-relación (E-R), que ofrece una manera de identificar las entidades que se van a representar en la base de datos y el modo en que se relacionan entre sí. Finalmente, el diseño de la base de datos se expresará en términos del diseño de bases de datos relationales y del conjunto de restricciones asociado. En este capítulo se mostrará la manera en que el diseño E-R puede transformarse en un conjunto de esquemas de relación y el modo en que se pueden incluir algunas de las restricciones en ese diseño. Luego, en el Capítulo 7, se considerará en detalle si el conjunto de esquemas de relación representa un diseño de la base de datos bueno o malo y se estudiará el proceso de creación de buenos diseños usando un conjunto de restricciones más amplio. Estos dos capítulos tratan los conceptos fundamentales del diseño de bases de datos.

6.1 Visión general del proceso de diseño

La tarea de creación de aplicaciones de bases de datos es una labor compleja, que implica varias fases, como el diseño del esquema de la base de datos, el diseño de los programas que tienen acceso a los datos y los actualizan y el diseño del esquema de seguridad para controlar el acceso a los datos. Las necesidades de los usuarios desempeñan un papel central en el proceso de diseño. En este capítulo nos centraremos en el diseño del esquema de la base de datos, aunque se esbozarán brevemente algunas de las otras tareas más adelante en este capítulo.

El diseño de un entorno completo de aplicaciones de bases de datos que responda a las necesidades de la empresa que se está modelando exige prestar atención a un amplio conjunto de consideraciones. Estos aspectos adicionales del uso esperado de la base de datos influyen en gran variedad de opciones de diseño en los niveles físico, lógico y de vistas.

6.1.1 Fases del diseño

Para aplicaciones pequeñas puede resultar factible para un diseñador de bases de datos que comprenda los requisitos de la aplicación decidir directamente sobre las relaciones que hay que crear, sus atributos y las restricciones sobre las relaciones. Sin embargo, un proceso de diseño tan directo resulta difícil para las aplicaciones reales, ya que a menudo son muy complejas. Frecuentemente no existe una sola persona que comprenda todas las necesidades de datos de la aplicación. El diseñador de la base de datos debe interactuar con los usuarios para comprender las necesidades de la aplicación, realizar una representación de alto nivel de esas necesidades que pueda ser comprendida por los usuarios y luego traducir esos requisitos a niveles inferiores del diseño. Los modelos de datos de alto nivel sirven a los diseñadores de bases de datos ofreciéndoles un marco conceptual en el que especificar de forma

sistemática los requisitos de datos de los usuarios de la base de datos y una estructura para la base de datos que satisfaga esos requisitos.

- La fase inicial del diseño de las bases de datos es la caracterización completa de las necesidades de datos de los posibles usuarios de la base de datos. El diseñador de la base de datos debe interactuar intensamente con los expertos y los usuarios del dominio para realizar esta tarea. El resultado de esta fase es una especificación de requisitos del usuario. Aunque existen técnicas para representar en diagramas los requisitos de los usuarios, en este capítulo sólo se describirán textualmente esos requisitos, que se ilustrarán posteriormente en el Apartado 6.8.2.
- A continuación, el diseñador elige el modelo de datos y, aplicando los conceptos del modelo de datos elegido, traduce estos requisitos en un esquema conceptual de la base de datos. El esquema desarrollado en esta fase de **diseño conceptual** proporciona una visión detallada de la empresa. Se suele emplear el modelo entidad-relación, que se estudiará en el resto de este capítulo, para representar el diseño conceptual. En términos del modelo entidad-relación, el esquema conceptual especifica las entidades que se representan en la base de datos, sus atributos, las relaciones entre ellas y las restricciones que las afectan. Generalmente, la fase de diseño conceptual da lugar a la creación de un diagrama entidad-relación que ofrece una representación gráfica del esquema.

El diseñador revisa el esquema para confirmar que realmente se satisfacen todos los requisitos y que no entran en conflicto entre sí. También puede examinar el diseño para eliminar características redundantes. Su atención en este momento se centra en describir los datos y sus relaciones, más que en especificar los detalles del almacenamiento físico.

- Un esquema conceptual completamente desarrollado indica también los requisitos funcionales de la empresa. En la **especificación de requisitos funcionales** los usuarios describen los tipos de operaciones (o transacciones) que se llevarán a cabo sobre los datos. Algunos ejemplos de operaciones son la modificación o actualización de datos, la búsqueda y recuperación de datos concretos y el borrado de datos. En esta fase de diseño conceptual el diseñador puede revisar el esquema para asegurarse de que satisface los requisitos funcionales.
- El proceso de paso desde el modelo abstracto de datos a la implementación de la base de datos se divide en dos fases de diseño finales.
 - En la **fase de diseño lógico** el diseñador traduce el esquema conceptual de alto nivel al modelo de datos de la implementación del sistema de bases de datos que se va a usar. El modelo de implementación de los datos suele ser el modelo relacional, y este paso suele consistir en la traducción del esquema conceptual definido mediante el modelo entidad-relación en un esquema de relación.
 - Finalmente, el diseñador usa el esquema de base de datos resultante propio del sistema en la siguiente **fase de diseño físico**, en la que se especifican las características físicas de la base de datos. Entre estas características están la forma de organización de los archivos y las estructuras de almacenamiento interno; se estudian en el Capítulo 11.

El esquema físico de la base de datos puede modificarse con relativa facilidad una vez creada la aplicación. Sin embargo, las modificaciones del esquema lógico suelen resultar más difíciles de llevar a cabo, ya que pueden afectar a varias consultas y actualizaciones dispersas por todo el código de la aplicación. Por tanto, es importante llevar a cabo con cuidado la fase de diseño de la base de datos antes de crear el resto de la aplicación de bases de datos.

6.1.2 Alternativas de diseño

Una parte importante del proceso de diseño de las bases de datos consiste en decidir la manera de representar en el diseño los diferentes tipos de “cosas”, como personas, lugares, productos y similares. Se usa el término *entidad* para hacer referencia a cualquiera de esos elementos claramente identificables. Las diferentes entidades tienen algunos aspectos en común y algunas diferencias. Se desea aprovechar los aspectos en común para tener un diseño conciso, fácilmente comprensible, pero hay que conservar la suficiente flexibilidad como para representar las diferencias entre las diferentes entidades existentes en

el momento del diseño o que se puedan materializar en el futuro. Las diferentes entidades se relacionan entre sí de diversas maneras, todas las cuales deben incluirse en el diseño de la base de datos.

Al diseñar el esquema de una base de datos hay que asegurarse de que se evitan dos peligros importantes:

1. **Redundancia.** Un mal diseño puede repetir información. En el ejemplo bancario que se ha venido usando, se tiene una relación con la información sobre los clientes y una relación separada con la información sobre las cuentas. Supóngase que, en lugar de eso, se repitiera toda la información sobre los clientes (nombre, dirección, etc.) una vez por cada cuenta o préstamo que tuviera cada cliente. Evidentemente, sería redundante. Lo ideal sería que la información apareciera exactamente en un solo lugar.
2. **Incompletitud.** Un mal diseño puede hacer que determinados aspectos de la empresa resulten difíciles o imposibles de modelar. Por ejemplo, supóngase que se usa un diseño de base de datos para el escenario bancario que almacena la información del nombre y de la dirección del cliente con cada cuenta y con cada préstamo, pero que no tiene una relación separada para los clientes. Resultaría imposible introducir el nombre y la dirección de los clientes nuevos a menos que ya tuvieran abierta una cuenta o concedido un préstamo. Se podría intentar salir del paso con este diseño problemático almacenando valores nulos para la información de las cuentas o de los préstamos, como puede ser el número de cuenta o el importe del préstamo. Este parche no sólo resulta poco atractivo, sino que puede evitarse mediante restricciones de clave primaria.

Evitar los malos diseños no es suficiente. Puede haber gran número de buenos diseños entre los que haya que escoger. Como ejemplo sencillo, considérese un cliente que compra un producto. ¿La venta de este producto es una relación entre el cliente y el producto? Dicho de otra manera, ¿es la propia venta una entidad que está relacionada con el cliente y con el producto? Esta elección, aunque simple, puede suponer una importante diferencia en cuanto a los aspectos de la empresa que se pueden modelar bien. Considerando la necesidad de tomar decisiones como ésta para el gran número de entidades y de relaciones que hay en las empresas reales, no es difícil ver que el diseño de bases de datos puede constituir un problema arduo. En realidad, se verá que exige una combinación de conocimientos y de “buen gusto”.

6.2 El modelo entidad-relación

El modelo de datos **entidad–relación** (E-R) se desarrolló para facilitar el diseño de bases de datos permitiendo la especificación de un *esquema de la empresa* que representa la estructura lógica global de la base de datos. El modelo de datos E-R es uno de los diferentes modelos de datos semánticos; el aspecto semántico del modelo radica en la representación del significado de los datos. El modelo E-R resulta muy útil para relacionar los significados e interacciones de las empresas reales con el esquema conceptual. Debido a esta utilidad, muchas herramientas de diseño de bases de datos se basan en los conceptos del modelo E-R. El modelo de datos E-R emplea tres conceptos básicos: los conjuntos de entidades, los conjuntos de relaciones y los atributos.

6.2.1 Conjuntos de entidades

Una **entidad** es una “cosa” u “objeto” del mundo real que es distingible de todos los demás objetos. Por ejemplo, cada persona de una empresa es una entidad. Una entidad tiene un conjunto de propiedades, y los valores de algún conjunto de propiedades pueden identificar cada entidad de forma única. Por ejemplo, el DNI 67.789.901 identifica únicamente una persona concreta de la empresa. Análogamente, los préstamos se pueden considerar entidades, y el número de préstamo P-15 de la sucursal de Navacerrada identifica únicamente una entidad de préstamo. Las entidades pueden ser concretas, como las personas o los libros, o abstractas, como los préstamos, las vacaciones o los conceptos.

Un **conjunto de entidades** es un conjunto de entidades del mismo tipo que comparten las mismas propiedades, o atributos. El conjunto de todas las personas que son clientes en un banco dado, por

32.112.312	Santos	Mayor	Peguerinos
1.928.374	Gómez	Carretas	Cerceda
67.789.901	López	Mayor	Peguerinos
55.555.555	Sotoca	Real	Cádiz
24.466.880	Pérez	Carretas	Cerceda
96.396.396	Valdivieso	Goya	Vigo
33.557.799	Fernández	Jazmín	León

P-17	1.000
P-23	2.000
P-15	1.500
P-14	1.500
P-19	500
P-11	900
P-16	1.300

cliente

préstamo

Figura 6.1 Conjuntos de entidades *cliente* y *préstamo*.

ejemplo, se puede definir como el conjunto de entidades *cliente*. Análogamente, el conjunto de entidades *préstamo* puede representar el conjunto de todos los préstamos concedidos por un banco concreto. Cada una de las entidades que constituyen un conjunto se denomina *extensión* de ese conjunto de entidades. Por tanto, todos los clientes de un banco son la extensión del conjunto de entidades *cliente*.

Los conjuntos de entidades no son necesariamente disjuntos. Por ejemplo, es posible definir el conjunto de entidades de todos los empleados de un banco (*empleado*) y el conjunto de entidades de todos los clientes del banco (*cliente*). Una entidad *persona* puede ser una entidad *empleado*, una entidad *cliente*, ambas cosas, o ninguna.

Cada entidad se representa mediante un conjunto de **atributos**. Los atributos son propiedades descriptivas que posee cada miembro de un conjunto de entidades. La designación de un atributo para un conjunto de entidades expresa que la base de datos almacena información parecida relativa a cada entidad del conjunto de entidades; sin embargo, cada entidad puede tener su propio valor para cada atributo. Posibles atributos del conjunto de entidades *cliente* son *id_cliente*, *nombre_cliente*, *calle_cliente* y *ciudad_cliente*. En la vida real habría más atributos, como el número de la calle, el número del piso la provincia, el código postal, y el país, pero se omiten para no complicar el ejemplo. Posibles atributos del conjunto de entidades *préstamo* son *número_préstamo* e *importe*.

Cada entidad tiene un **valor** para cada uno de sus atributos. Por ejemplo, una entidad *cliente* concreta puede tener el valor 32.112.312 para *id_cliente*, el valor Santos para *nombre_cliente*, el valor Mayor para *calle_cliente* y el valor Peguerinos para *ciudad_cliente*.

El atributo *id_cliente* se usa para identificar únicamente a los clientes, dado que puede haber más de un cliente con el mismo nombre, calle y ciudad. En Estados Unidos, muchas empresas consideran adecuado usar el número *seguridad_social* de cada persona¹ como atributo cuyo valor identifica únicamente a esa persona. En general la empresa tendría que crear y asignar un identificador único a cada cliente.

Por tanto, las bases de datos incluyen una serie de conjuntos de entidades, cada una de las cuales contiene cierto número de entidades del mismo tipo. La Figura 6.1 muestra parte de una base de datos de un banco que consta de dos conjuntos de entidades, *cliente* y *préstamo*.

Las bases de datos para entidades bancarias pueden incluir diferentes conjuntos de entidades. Por ejemplo, además del seguimiento de los clientes y de los préstamos, el banco también ofrece cuentas, que se representan mediante el conjunto de entidades *cuenta* con los atributos *número_cuenta* y *saldo*. Además, si el banco tiene varias sucursales, se puede guardar información acerca de todas las sucursales del

1. En España se asigna a cada ciudadano un número único, denominado número del documento nacional de identidad (DNI) para identificarla únicamente. Se supone que cada persona tiene un único DNI y que no hay dos personas con el mismo DNI.

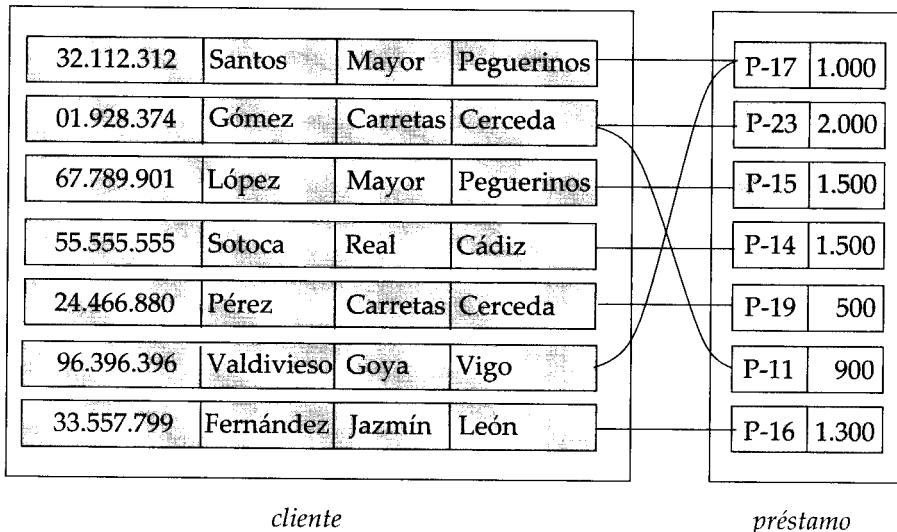


Figura 6.2 Conjunto de relaciones *prestatario*.

banco. Cada conjunto de entidades *sucursal* se puede describir mediante los atributos *nombre_sucursal*, *ciudad_sucursal* y *activos*.

6.2.2 Conjuntos de relaciones

Una **relación** es una asociación entre varias entidades. Por ejemplo, se puede definir una relación que asocie al cliente López con el préstamo P-15. Esta relación especifica que López es un cliente con el préstamo número P-15.

Un **conjunto de relaciones** es un conjunto de relaciones del mismo tipo. Formalmente es una relación matemática con $n \geq 2$ de conjuntos de entidades (posiblemente no distintos). Si E_1, E_2, \dots, E_n son conjuntos de entidades, entonces un conjunto de relaciones R es un subconjunto de:

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

donde (e_1, e_2, \dots, e_n) es una relación.

Considérense las dos entidades *cliente* y *préstamo* de la Figura 6.1. Se define el conjunto de relaciones *prestatario* para denotar la asociación entre los clientes y los préstamos bancarios que tienen concedidos. La Figura 6.2 muestra esta asociación.

Como ejemplo adicional, considérense los dos conjuntos de entidades *préstamo* y *sucursal*. Se puede definir el conjunto de relaciones *sucursal_préstamo* para denotar la asociación entre un préstamo y la sucursal en que se concede ese préstamo.

La asociación entre conjuntos de entidades se conoce como *participación*; es decir, los conjuntos de entidades E_1, E_2, \dots, E_n **participan** en el conjunto de relaciones R. Un **ejemplar de la relación** de un esquema E-R representa una asociación entre las entidades citadas en la empresa real que se está modelando. Como ilustración, la entidad *cliente* López, que tiene el identificador de cliente 67.789.901, y la entidad *préstamo* P-15 participan en un ejemplar de la relación *prestatario*. Este ejemplar de relación representa que, en la empresa real, la persona llamada López que tiene el *id_cliente* 67.789.901 tiene concedido el préstamo que está numerado como P-15.

La función que desempeña una entidad en una relación se denomina **rol** de esa entidad. Como los conjuntos de entidades que participan en un conjunto de relaciones, generalmente, son distintos, los roles están implícitos y no se suelen especificar. Sin embargo, resultan útiles cuando el significado de una relación necesita aclaración. Tal es el caso cuando los conjuntos de entidades de una relación no son distintos; es decir, el mismo conjunto de entidades participa en un conjunto de relaciones más de una vez, con diferentes roles. En este tipo de conjunto de relaciones, que se denomina a veces conjunto de relaciones **recursivo**, son necesarios los nombres explícitos para los roles para especificar la manera

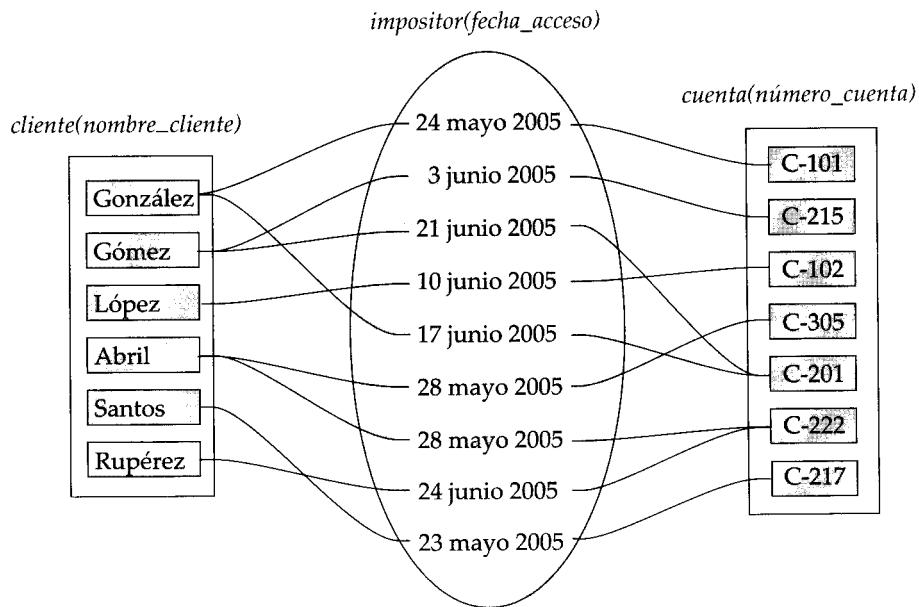


Figura 6.3 Fecha_acceso como atributo del conjunto de entidades *impositor*.

en que cada entidad participa en cada ejemplar de la relación. Por ejemplo, considérese un conjunto de entidades *empleado* que almacena información sobre todos los empleados del banco. Se puede tener un conjunto de relaciones *trabaja_para* que se modela mediante pares ordenados de entidades *empleado*. El primer empleado de cada par adopta el rol de *trabajador*, mientras el segundo desempeña el rol de *jefe*. De esta manera, todas las relaciones *trabaja_para* se caracterizan mediante pares (*trabajador, jefe*); los pares (*jefe, trabajador*) quedan excluidos.

Una relación puede también tener atributos denominados **atributos descriptivos**. Considérese el conjunto de relaciones *impositor* con los conjuntos de entidades *cliente* y *cuenta*. Se puede asociar el atributo *fecha_acceso* con esta relación para especificar la fecha más reciente de acceso del cliente a la cuenta. La relación *impositor* entre las entidades correspondientes al cliente Santos y la cuenta C-217 tiene el valor “23 mayo 2005” para el atributo *fecha_acceso*, lo que significa que el último día en que Santos accedió a la cuenta C-217 fue el 23 de mayo de 2005.

La Figura 6.3 muestra el conjunto de relaciones *impositor* con el atributo descriptivo *fecha_acceso*; en aras de la sencillez, sólo se muestran algunos atributos de los dos conjuntos de entidades.

Como ejemplo adicional de los atributos descriptivos de las relaciones, supóngase que se tienen los conjuntos de entidades *estudiante* y *asignatura* que participan en el conjunto de relaciones *matriculado*. Puede que se desee almacenar el atributo descriptivo *créditos* con la relación, para registrar si el estudiante se ha matriculado en la asignatura para obtener créditos o sólo como oyente.

Cada ejemplar de una relación de un conjunto de relaciones determinado debe identificarse únicamente a partir de sus entidades participantes, sin usar los atributos descriptivos. Para comprender esto, supóngase que deseemos representar todas las fechas en las que un cliente ha tenido acceso a una cuenta. El atributo monovalorado *fecha_acceso* sólo puede almacenar una fecha de acceso. No se pueden representar varias fechas de acceso mediante varios ejemplares de la relación entre el mismo cliente y la misma cuenta, ya que los ejemplares de la relación no quedarían identificados únicamente usando solamente las entidades participantes. La forma correcta de tratar este caso es crear el atributo multivalorado *fechas_acceso*, que puede almacenar todas las fechas de acceso.

Sin embargo, puede haber más de un conjunto de relaciones que implique a los mismos conjuntos de entidades. En nuestro ejemplo, los conjuntos de entidades *cliente* y *préstamo* participan en el conjunto de relaciones *prestatario*. Además, supóngase que cada préstamo deba tener otro cliente que sirva como avalista del préstamo. Entonces los conjuntos de entidades *cliente* y *préstamo* pueden participar en otro conjunto de relaciones: *avalista*.

Los conjuntos de relaciones *prestatario* y *sucursal_préstamo* proporcionan un ejemplo de conjunto de relaciones **binario**—es decir, uno que implica dos conjuntos de entidades. La mayor parte de los conjuntos de relaciones de los sistemas de bases de datos son binarios. A veces, no obstante, los conjuntos de relaciones implican a más de dos conjuntos de entidades.

Por ejemplo, considérense los conjuntos de entidades *empleado*, *sucursal* y *trabajo*. Ejemplos de entidades *trabajo* pueden ser director, cajero, interventor, etc. Las entidades *trabajo* pueden tener los atributos *puesto* y *nivel*. El conjunto de relaciones *trabaja_en* entre *empleado*, *sucursal* y *trabajo* es un ejemplo de relación ternaria. Una relación ternaria entre Santos, Navacerrada y director indica que Santos trabaja de director de la sucursal de Navacerrada. Santos también podría actuar como interventor de la sucursal de Centro, lo que estaría representado por otra relación. Podría haber otra relación entre Gómez, Centro y cajero, que indicaría que Gómez trabaja de cajero en la sucursal de Centro.

El número de conjuntos de entidades que participan en un conjunto de relaciones es también el **grado** de ese conjunto de relaciones. Los conjuntos de relaciones binarios tienen grado 2; los conjuntos de relaciones ternarios tienen grado 3.

6.2.3 Atributos

Para cada atributo hay un conjunto de valores permitidos, denominados **dominio** o **conjunto de valores** de ese atributo. El dominio del atributo *nombre_cliente* puede ser el conjunto de todas las cadenas de texto de una cierta longitud. Análogamente, el dominio del atributo *número_préstamo* puede ser el conjunto de todas las cadenas de caracteres de la forma “P-*n*”, donde *n* es un entero positivo.

Formalmente, cada atributo de un conjunto de entidades es una función que asigna el conjunto de entidades a un dominio. Dado que el conjunto de entidades puede tener varios atributos, cada entidad se puede describir mediante un conjunto de pares (atributo, valor), un par por cada atributo del conjunto de entidades. Por ejemplo, una entidad *cliente* concreta se puede describir mediante el conjunto (*id_cliente*, 67.789.901), (*nombre_cliente*, López), (*calle_cliente*, Mayor), (*ciudad_cliente*, Peguerinos), lo que significa que esa entidad describe a una persona llamada López que tiene el DNI número 67.789.901 y que reside en la calle Mayor de Peguerinos. Ahora se puede ver que existe una integración del esquema abstracto con la empresa real que se está modelando. Los valores de los atributos que describen cada entidad constituyen una parte significativa de los datos almacenados en la base de datos.

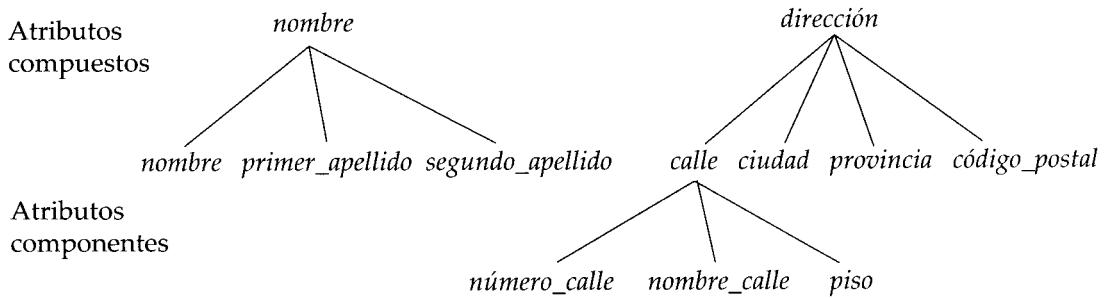
Cada atributo, tal y como se usa en el modelo E-R, se puede caracterizar por los siguientes tipos de atributo.

- **Atributos simples y compuestos.** En los ejemplos considerados hasta ahora los atributos han sido simples; es decir, no estaban divididos en subpartes. Los atributos **compuestos**, en cambio, se pueden dividir en subpartes (es decir, en otros atributos). Por ejemplo, el atributo *nombre* puede estar estructurado como un atributo compuesto consistente en *nombre*, *primer_apellido* y *segundo_apellido*. Usar atributos compuestos en un esquema de diseño es una buena elección si el usuario desea referirse a un atributo completo en algunas ocasiones y, en otras, solamente a algún componente del atributo. Supóngase que hubiera que sustituir para el conjunto de entidades *cliente* los atributos *calle_cliente* y *ciudad_cliente* por el atributo compuesto *dirección*, con los atributos *calle*, *ciudad*, *provincia*, y *código_postal*². Los atributos compuestos ayudan a agrupar atributos relacionados, lo que hace que los modelos sean más claros.

Obsérvese también que los atributos compuestos pueden aparecer como una jerarquía. En el atributo compuesto *dirección*, el componente *calle* puede dividirse a su vez en *número_calle*, *nombre_calle* y *piso*. La Figura 6.4 muestra estos ejemplos de atributos compuestos para el conjunto de entidades *cliente*.

- **Atributos monovalorados y multivalorados.** Todos los atributos que se han especificado en los ejemplos anteriores tienen un único valor para cada entidad concreta. Por ejemplo, el atributo *número_préstamo* para una entidad *préstamo* específica hace referencia a un único número de préstamo. Se dice que estos atributos son **monovalorados**. Puede haber ocasiones en las que

2. Se asume el formato de *calle_cliente* y *dirección* usado en España, que incluye un código postal numérico llamado “código postal”.

**Figura 6.4** Atributos compuestos *nombre* y *dirección*.

un atributo tenga un conjunto de valores para una entidad concreta. Considérese un conjunto de entidades *empleado* con el atributo *número_telefón*. Cada empleado puede tener cero, uno o varios números de teléfono, y empleados diferentes pueden tener diferente cantidad de teléfonos. Se dice que este tipo de atributo es **multivalorado**. Como ejemplo adicional, el atributo *nombre_subordinado* del conjunto de entidades *empleado* es multivalorado, ya que cada empleado podría tener cero, uno o más subordinados.

Si resulta necesario, se pueden establecer apropiadamente límites inferior y superior al número de valores en el atributo **multivalorado**. Por ejemplo, el banco puede limitar a dos el número de números de teléfono que se guardan por cliente. El establecimiento de límites en este caso expresa que el atributo *número_telefón* del conjunto de entidades *cliente* puede tener entre cero y dos valores.

- **Atributos derivados.** El valor de este tipo de atributo se puede obtener a partir del valor de otros atributos o entidades relacionados. Por ejemplo, supóngase que el conjunto de entidades *cliente* que tiene un atributo *préstamos* que representa el número de préstamos que cada cliente tiene concedidos en el banco. Ese atributo se puede obtener contando el número de entidades *préstamo* asociadas con cada cliente.

Como ejemplo adicional, supóngase que el conjunto de entidades *cliente* tiene el atributo *edad*, que indica la edad del cliente. Si el conjunto de entidades *cliente* tiene también un atributo *fecha_de_nacimiento*, se puede calcular *edad* a partir de *fecha_de_nacimiento* y de la fecha actual. Por tanto, *edad* es un atributo derivado. En este caso, *fecha_de_nacimiento* puede considerarse un atributo **básico**, o *almacenado*. El valor de los atributos derivados no se almacena, sino que se calcula cada vez que hace falta.

Los atributos toman valores **nulos** cuando las entidades no tienen ningún valor para ese atributo. El valor *nulo* también puede indicar “no aplicable”—es decir, que el valor no existe para esa entidad. Por ejemplo, una persona puede no tener un segundo nombre de pila. *Nulo* puede también designar que el valor del atributo es desconocido. Un valor desconocido puede ser *falta* (el valor existe pero no se tiene esa información) o *desconocido* (no se sabe si ese valor existe realmente o no).

Por ejemplo, si el valor *nombre* de un *cliente* dado es *nulo*, se da por supuesto que el valor es *falta*, ya que todos los clientes deben tener nombre. Un valor *nulo* para el atributo *piso* puede significar que la dirección no incluye un piso (no aplicable), que existe el valor piso pero no se conoce cuál es (*falta*), o que no se sabe si el valor piso forma parte o no de la dirección del cliente (*desconocido*).

6.3 Restricciones

Un esquema de desarrollo E-R puede definir ciertas restricciones a las que el contenido de la base de datos se debe adaptar. En este apartado se examinan la correspondencia de cardinalidades, las restricciones de claves y las restricciones de participación.

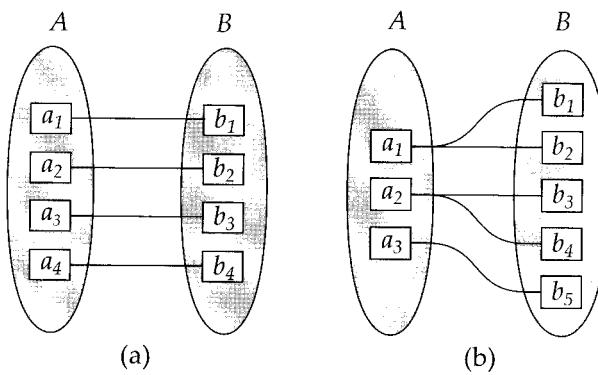


Figura 6.5 Correspondencia de cardinalidades. (a) Uno a uno. (b) Uno a varios.

6.3.1 Correspondencia de cardinalidades

La **correspondencia de cardinalidades**, o razón de cardinalidad, expresa el número de entidades a las que otra entidad se puede asociar mediante un conjunto de relaciones.

La correspondencia de cardinalidades resulta muy útil para describir conjuntos de relaciones binarias, aunque pueda contribuir a la descripción de conjuntos de relaciones que impliquen más de dos conjuntos de entidades. En este apartado se centrará la atención únicamente en los conjuntos de relaciones binarias.

Para un conjunto de relaciones binarias R entre los conjuntos de entidades A y B , la correspondencia de cardinalidades debe ser una de las siguientes:

- **Uno a uno** Cada entidad de A se asocia, *a lo sumo*, con una entidad de B , y cada entidad en B se asocia, *a lo sumo*, con una entidad de A (véase la Figura 6.5a).
- **Uno a varios** Cada entidad de A se asocia con cualquier número (cero o más) de entidades de B . Cada entidad de B , sin embargo, se puede asociar, *a lo sumo*, con una entidad de A (véase la Figura 6.5b).
- **Varios a uno** Cada entidad de A se asocia, *a lo sumo*, con una entidad de B . Cada entidad de B , sin embargo, se puede asociar con cualquier número (cero o más) de entidades de A (véase la Figura 6.6a).
- **Varios a varios** Cada entidad de A se asocia con cualquier número (cero o más) de entidades de B , y cada entidad de B se asocia con cualquier número (cero o más) de entidades de A (véase la Figura 6.6b).

La correspondencia de cardinalidades adecuada para un conjunto de relaciones dado depende, obviamente, de la situación del mundo real que el conjunto de relaciones modele.

Como ilustración, considérese el conjunto de relaciones *prestatario*. Si, en un banco dado, cada préstamo sólo puede pertenecer a un cliente y cada cliente puede tener varios préstamos, entonces el conjunto de relaciones de *cliente a préstamo* es uno a varios. Si cada préstamo puede pertenecer a varios clientes (como los préstamos solicitados conjuntamente por varios socios de un negocio) el conjunto de relaciones es varios a varios. La Figura 6.2 muestra este tipo de relación.

6.3.2 Claves

Es necesario tener una forma de especificar la manera de distinguir las entidades pertenecientes a un conjunto de entidades dado. Conceptualmente cada entidad es distinta; desde el punto de vista de las bases de datos, sin embargo, la diferencia entre ellas se debe expresar en términos de sus atributos.

Por lo tanto, los valores de los atributos de cada entidad deben ser tales que permitan *identificar únicamente* a esa entidad. En otras palabras, no se permite que ningún par de entidades de un conjunto de entidades tenga exactamente el mismo valor en todos sus atributos.

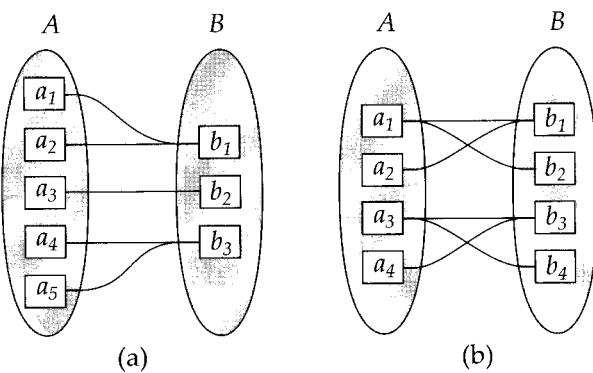


Figura 6.6 Correspondencia de cardinalidades. (a) Varios a uno. (b) Varios a varios.

Las *claves* permiten identificar un conjunto de atributos que resulta suficiente para distinguir las entidades entre sí. Las claves también ayudan a identificar únicamente las relaciones y, por tanto, a distinguir las relaciones entre sí.

6.3.2.1 Conjuntos de entidades

Una **superclave** es un conjunto de uno o más atributos que, tomados conjuntamente, permiten identificar de forma única una entidad del conjunto de entidades. Por ejemplo, el atributo *id_cliente* del conjunto de entidades *cliente* es suficiente para distinguir una entidad *cliente* de las demás. Por tanto, *id_cliente* es una superclave. Análogamente, la combinación de *nombre_cliente* e *id_cliente* es una superclave del conjunto de entidades *cliente*. El atributo *nombre_cliente* de *cliente* no es una superclave, ya que varias personas pueden tener el mismo nombre.

El concepto de superclave no es suficiente para lo que aquí se propone ya que, como se ha visto, las superclaves pueden contener atributos innecesarios. Si C es una superclave, entonces también lo es cualquier superconjunto de C. A menudo interesan las superclaves para las que ningún subconjunto propio es superclave. Esas superclaves mínimas se denominan **claves candidatas**.

Es posible que conjuntos distintos de atributos puedan servir como clave candidata. Supóngase que una combinación de *nombre_cliente* y *calle_cliente* sea suficiente para distinguir entre los miembros del conjunto de entidades *cliente*. Entonces, tanto *id_cliente* como *nombre_cliente*, *calle_cliente* son claves candidatas. Aunque los atributos *id_cliente* y *nombre_cliente* juntos puedan diferenciar las entidades *cliente*, su combinación no forma una clave candidata, ya que el atributo *id_cliente* por sí solo es una clave candidata.

Se usa el término **clave primaria** para denotar la clave candidata elegida por el diseñador de la base de datos como elemento principal de identificación de las entidades pertenecientes a un conjunto de entidades. Las claves (primarias, candidatas y superclaves) son propiedades del conjunto de entidades, más que de cada una de las entidades. Dos entidades cualesquiera del conjunto no pueden tener el mismo valor de los atributos de su clave al mismo tiempo. La designación de una clave representa una restricción de la empresa real que se está modelando.

Las claves candidatas se deben escoger con cuidado. Como se ha indicado, el nombre de cada persona es obviamente insuficiente, ya que puede haber muchas personas con el mismo nombre. En España, el atributo DNI de cada persona puede ser una clave candidata. Como los no ciudadanos de en España no tienen DNI, las empresas con trabajadores extranjeros deben generar sus propios identificadores únicos. Una alternativa es usar como clave alguna combinación única de otros atributos.

La clave primaria se debe escoger de manera que sus atributos nunca, o muy raramente, cambien. Por ejemplo, el campo dirección de cada persona no debe formar parte de la clave primaria, ya que es probable que cambie. El número de DNI, por otra parte, es seguro que no cambiará. Los identificadores únicos generados por las empresas generalmente no cambian, excepto si se fusionan dos empresas; en tal caso, las dos empresas pueden haber emitido el mismo identificador y será necesario reasignar los identificadores para asegurarse de que sean únicos.

6.3.2.2 Conjuntos de relaciones

La clave primaria de cada conjunto de entidades permite distinguir entre las diferentes entidades del conjunto. Se necesita un mecanismo parecido para distinguir entre las diferentes relaciones de cada conjunto de relaciones.

Sea R un conjunto de relaciones que involucra los conjuntos de entidades E_1, E_2, \dots, E_n . Sea $clave_primaria(E_i)$ el conjunto de atributos que forman la clave primaria del conjunto de entidades E_i . Por ahora se dará por supuesto que los nombres de los atributos de todas las claves primarias son únicos y que cada conjunto de entidades participa sólo una vez en la relación. La composición de la clave primaria de un conjunto de relaciones depende del conjunto de atributos asociado con el conjunto de relaciones R .

Si el conjunto de relaciones R no tiene atributos asociados, entonces el conjunto de atributos

$$clave_primaria(E_1) \cup clave_primaria(E_2) \cup \dots \cup clave_primaria(E_n)$$

describe una relación concreta del conjunto R .

Si el conjunto de relaciones R tiene asociados los atributos a_1, a_2, \dots, a_m , entonces el conjunto de atributos

$$clave_primaria(E_1) \cup clave_primaria(E_2) \cup \dots \cup clave_primaria(E_n) \cup \{a_1, a_2, \dots, a_m\}$$

describe una relación concreta del conjunto R .

En ambos casos, el conjunto de atributos

$$clave_primaria(E_1) \cup clave_primaria(E_2) \cup \dots \cup clave_primaria(E_n)$$

forma una superclave del conjunto de relaciones.

En el caso de que los nombres de los atributos de las claves primarias no sean únicos en todos los conjuntos de entidades, hay que renombrar los atributos para distinguirlos; el nombre del conjunto de entidades combinado con el del atributo formará un nombre único. En el caso de que un conjunto de entidades participe más de una vez en el conjunto de relaciones (como en la relación *trabaja_para* del Apartado 6.2.2), se usa el nombre del rol en lugar del nombre del conjunto de entidades para formar un nombre de atributo único.

La estructura de la clave primaria para el conjunto de relaciones depende de la correspondencia de cardinalidades del conjunto de relaciones. Como ejemplo, considérense los conjuntos de entidades *cliente* y *cuenta* y el conjunto de relaciones *impositor*, con el atributo *fecha_acceso* del Apartado 6.2.2. Supóngase que el conjunto de relaciones es varios a varios. Entonces, la clave primaria de *impositor* consiste en la unión de las claves primarias de *cliente* y de *cuenta*. Sin embargo, si un cliente sólo puede tener una cuenta—es decir, si la relación *impositor* es varios a uno de *cliente* a *cuenta*—entonces la clave primaria de *impositor* es simplemente la clave primaria de *cliente*. Análogamente, si la relación es varios a uno de *cuenta* a *cliente*—es decir, cada cuenta pertenece, a lo sumo, a un cliente—entonces la clave primaria de *impositor* es simplemente la clave primaria de *cuenta*. Para relaciones uno a uno se puede usar cualquiera de las claves primarias.

Para las relaciones no binarias, si no hay restricciones de cardinalidad, la superclave formada como se ha descrito anteriormente en este apartado es la única clave candidata, y se elige como clave primaria. La elección de la clave primaria resulta más complicada si hay restricciones de cardinalidad. Dado que no se ha estudiado la manera de especificar las restricciones de cardinalidad en relaciones no binarias, no se estudiará más este aspecto en este capítulo. Se considerará este aspecto con más detalle en el Apartado 7.4.

6.3.3 Restricciones de participación

Se dice que la participación de un conjunto de entidades E en un conjunto de relaciones R es **total** si cada entidad de E participa, al menos, en una relación de R . Si sólo algunas entidades de E participan en relaciones de R , se dice que la participación del conjunto de entidades E en la relación R es **parcial**. Por ejemplo, se puede esperar que cada entidad *préstamo* esté relacionada, al menos, con un cliente mediante

la relación *prestatario*. Por tanto, la participación de *préstamo* en el conjunto de relaciones *prestatario* es total. En cambio, un individuo puede ser cliente del banco tenga o no tenga concedido algún préstamo en el banco. Por tanto, es posible que sólo algunas de las entidades *cliente* estén relacionadas con el conjunto de entidades *préstamo* mediante la relación *prestatario*, y la participación de *cliente* en la relación *prestatario* es, por tanto, parcial.

6.4 Diagramas entidad-relación

Como se vio brevemente en el Apartado 1.3.3, los **diagramas E-R** pueden expresar gráficamente la estructura lógica general de las bases de datos. Los diagramas E-R son sencillos y claros—cualidades que pueden ser responsables en gran parte de la popularidad del modelo E-R. Estos diagramas constan de los siguientes componentes principales:

- **Rectángulos**, que representan conjuntos de entidades.
- **Elipses**, que representan atributos.
- **Rombos**, que representan conjuntos de relaciones.
- **Líneas**, que unen los atributos con los conjuntos de entidades y los conjuntos de entidades con los conjuntos de relaciones.
- **Elipses dobles**, que representan atributos multivalorados.
- **Elipses discontinuas**, que denotan atributos derivados.
- **Líneas dobles**, que indican participación total de una entidad en un conjunto de relaciones.
- **Rectángulos dobles**, que representan conjuntos de entidades débiles (se describirán posteriormente en el Apartado 6.6).

Considérese el diagrama entidad-relación de la Figura 6.7, que consiste en dos conjuntos de entidades, *cliente* y *préstamo*, relacionados mediante el conjunto de relaciones binarias *prestatario*. Los atributos asociados con *cliente* son *id_cliente*, *nombre_cliente*, *calle_cliente*, y *ciudad_cliente*. Los atributos asociados con *préstamo* son *número_préstamo* e *importe*. En la Figura 6.7 los atributos del conjunto de entidades que son miembros de la clave primaria están subrayados.

El conjunto de relaciones *prestatario* puede ser varios a varios, uno a varios, varios a uno o uno a uno. Para distinguir entre estos tipos, se dibuja o una línea dirigida (\rightarrow) o una línea no dirigida ($-$) entre el conjunto de relaciones y el conjunto de entidades en cuestión.

- Una línea dirigida desde el conjunto de relaciones *prestatario* al conjunto de entidades *préstamo* especifica que *prestatario* es un conjunto de relaciones uno a uno o varios a uno desde *cliente* a *préstamo*; *prestatario* no puede ser un conjunto de relaciones varios a varios ni uno a varios desde *cliente* a *préstamo*.

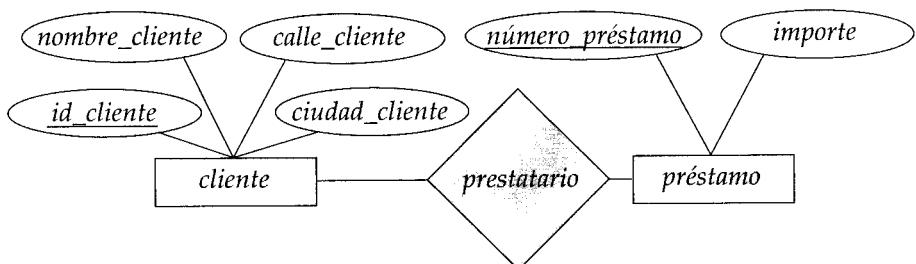


Figura 6.7 Diagrama E-R correspondiente a clientes y préstamos.

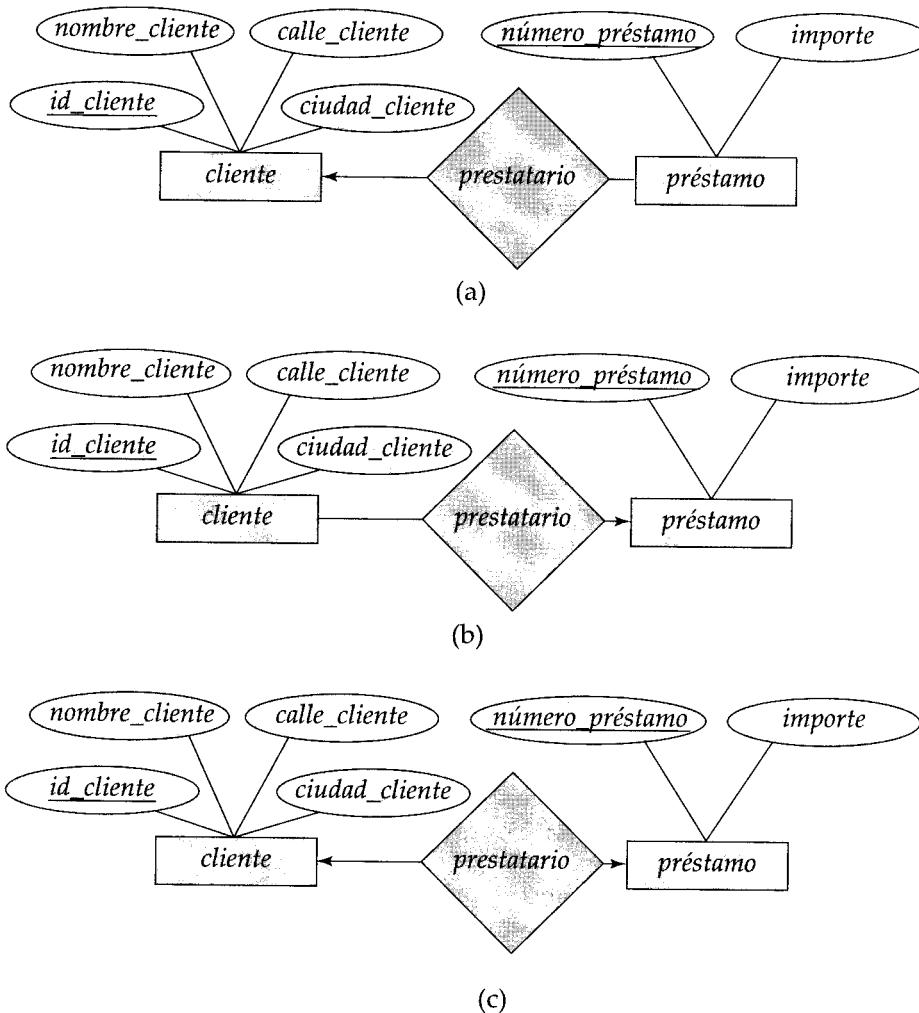


Figura 6.8 Relaciones. (a) Uno a varios. (b) Varios a uno. (c) Uno a uno.

- Una línea no dirigida desde el conjunto de relaciones *prestatario* al conjunto de relaciones *préstamo* especifica que *prestatario* es un conjunto de relaciones varios a varios o uno a varios desde *cliente* a *préstamo*.

Volviendo al diagrama E-R de la Figura 6.7, puede verse que el conjunto de relaciones *prestatario* es varios a varios. Si el conjunto de relaciones *prestatario* fuera uno a varios, desde *cliente* a *préstamo*, entonces la línea desde *prestatario* a *cliente* sería dirigida, con una flecha que apuntaría al conjunto de

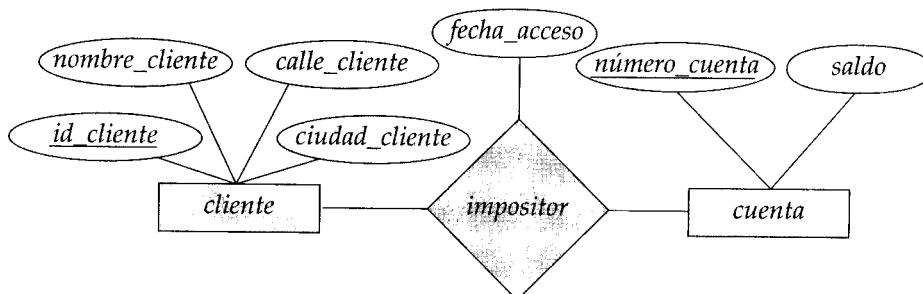


Figura 6.9 Diagrama E-R con un atributo unido a un conjunto de relaciones.

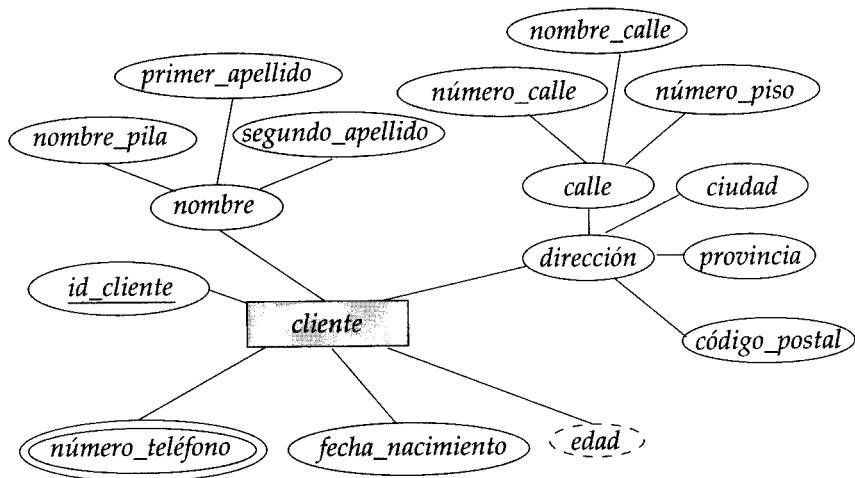


Figura 6.10 Diagrama E-R con atributos compuestos, multivalorados y derivados.

entidades *cliente* (Figura 6.8a). Análogamente, si el conjunto de relaciones *prestatario* fuera varios a uno desde *cliente* a *préstamo*, entonces la línea desde *prestatario* a *préstamo* tendría una flecha que apuntaría al conjunto de entidades *préstamo* (Figura 6.8b). Finalmente, si el conjunto de relaciones *prestatario* fuera uno a uno, entonces las dos líneas que salen de *prestatario* tendrían flecha: una que apuntaría al conjunto de entidades *préstamo* y otra que apuntaría al conjunto de entidades *cliente* (Figura 6.8c).

Si un conjunto de relaciones también tiene asociados algunos atributos, entonces esos atributos se unen con el conjunto de relaciones. Por ejemplo, en la Figura 6.9, se tiene el atributo descriptivo *fecha_acceso* unido al conjunto de relaciones *impositor* para especificar la fecha del último acceso del cliente a esa cuenta.

La Figura 6.10 muestra cómo se pueden representar los atributos compuestos en la notación E-R. En este caso, el atributo compuesto *nombre*, con los atributos componentes *nombre_pila*, *primer_apellido* y *segundo_apellido* sustituye al atributo simple *nombre_cliente* de *cliente*. Además, el atributo compuesto *dirección*, cuyos atributos componentes son *calle*, *ciudad*, *provincia* y *código_postal*, sustituye a los atributos *calle_cliente* y *ciudad_cliente* de *cliente*. El atributo *calle* es por sí mismo un atributo compuesto cuyos atributos componentes son *número_calle*, *nombre_calle* y *número_piso*.

La Figura 6.10 también muestra un atributo multivalorado, *numero_telefono*, indicado por una elipse doble, y un atributo derivado, *edad*, indicado por una elipse discontinua.

En los diagramas E-R los roles se indican mediante etiquetas en las líneas que unen los rombos con los rectángulos. La Figura 6.11 muestra los indicadores de roles *director* y *trabajador* entre el conjunto de entidades *empleado* y el conjunto de relaciones *trabaja_para*.

Los conjuntos de relaciones no binarias se pueden especificar fácilmente en los diagramas E-R. La Figura 6.12 consta de tres conjuntos de entidades *empleado*, *trabajo* y *sucursal*, relacionados mediante el conjunto de relaciones *trabaja_en*.

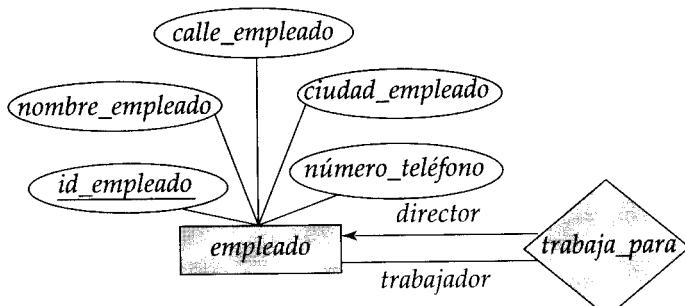


Figura 6.11 Diagrama E-R con indicadores de roles.

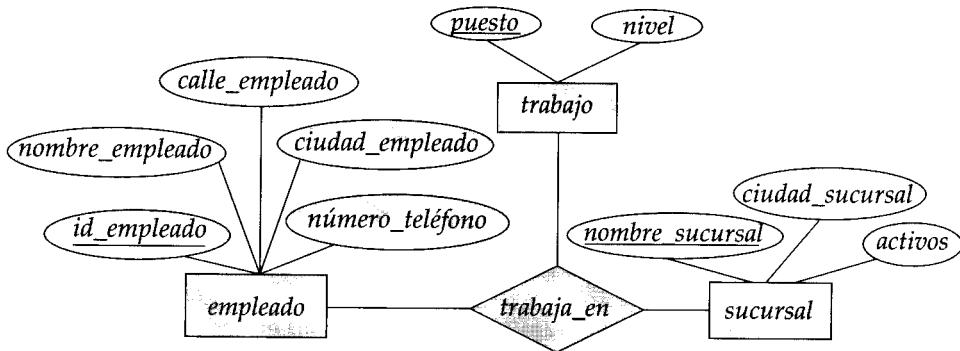


Figura 6.12 Diagrama E-R con una relación ternaria.

En el caso de conjuntos de relaciones no binarias se pueden especificar algunos tipos de relaciones varios a uno. Supóngase que un empleado puede tener, a lo sumo, un trabajo en cada sucursal (por ejemplo, Santos no puede ser director e interventor en la misma sucursal). Esta restricción se puede especificar mediante una flecha que apunte a **trabajo** en el arco de **trabaja_en**.

Como máximo se permite una flecha desde cada conjunto de relaciones, ya que los diagramas E-R con dos o más flechas salientes de cada conjunto de relaciones no binarias se pueden interpretar de dos formas. Supónganse que hay un conjunto de relaciones R entre los conjuntos de entidades A_1, A_2, \dots, A_n y que las únicas flechas están en los arcos de los conjuntos de entidades $A_{i+1}, A_{i+2}, \dots, A_n$. Entonces, las dos interpretaciones posibles son:

1. Una combinación concreta de entidades de A_1, A_2, \dots, A_i se puede asociar, a lo sumo, con una combinación de entidades de $A_{i+1}, A_{i+2}, \dots, A_n$. Por tanto, la clave primaria de la relación R se puede crear mediante la unión de las claves primarias de A_1, A_2, \dots, A_i .
2. Para cada conjunto de entidades A_k , $i < k \leq n$, cada combinación de las entidades de los otros conjuntos de entidades se puede asociar, a lo sumo, con una entidad de A_k . Cada conjunto $\{A_1, A_2, \dots, A_{k-1}, A_{k+1}, \dots, A_n\}$, para $i < k \leq n$, forma, entonces, una clave candidata.

Cada una de estas interpretaciones se ha usado en diferentes libros y sistemas. Para evitar confusiones, sólo se permite una flecha saliente de cada conjunto de relaciones, en cuyo caso las dos interpretaciones son equivalentes. En el Capítulo 7 (Apartado 7.4) se estudia el concepto de *dependencia funcional*, que permite especificar cualquiera de estas dos interpretaciones sin ambigüedad.

En los diagramas E-R se usan líneas dobles para indicar que la participación de un conjunto de entidades en un conjunto de relaciones es total; es decir, cada entidad del conjunto de entidades aparece, al menos, en una relación de ese conjunto de relaciones. Por ejemplo, considérese la relación *prestatario* entre clientes y préstamos. Una línea doble de *préstamo a prestatario*, como en la Figura 6.13, indica que cada préstamo debe tener, al menos, un cliente asociado.

Los diagramas E-R también ofrecen una manera de indicar restricciones más complejas sobre el número de veces que cada entidad participa en las relaciones de un conjunto de relaciones. Un segmento entre un conjunto de entidades y un conjunto de relaciones binarias puede tener unas cardinalidades

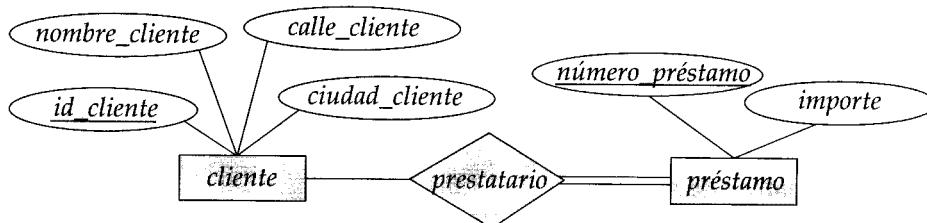


Figura 6.13 Participación total de un conjunto de entidades en un conjunto de relaciones.

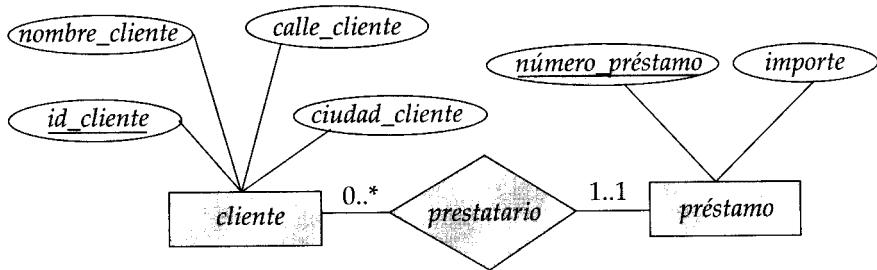


Figura 6.14 Límites de cardinalidad en los conjuntos de relaciones.

mínima y máxima asociadas , que se muestran de la forma *min..max*, donde *min* es la cardinalidad mínima y *max* es la máxima. Un valor mínimo de 1 indica una participación total del conjunto de entidades en el conjunto de relaciones. Un valor máximo de 1 indica que la entidad participa, a lo sumo, en una relación, mientras que un valor máximo de * indica que no hay límite. Téngase en cuenta que una etiqueta 1..* en un segmento es equivalente a una línea doble.

Por ejemplo, considérese la Figura 6.14. El segmento entre *préstamo* y *prestatario* tiene una restricción de cardinalidad de 1..1, que significa que tanto la cardinalidad mínima como la máxima son 1. Es decir, cada préstamo debe tener exactamente un cliente asociado. El límite 0..* en el segmento de *cliente* a *prestatario* indica que cada cliente puede no tener ningún préstamo o tener varios. Por tanto, la relación *prestatario* es uno a varios de *cliente* a *préstamo* y, además, la participación de *préstamo* en *prestatario* es total.

Es fácil malinterpretar 0..* en el segmento entre *cliente* y *prestatario* y pensar que la relación *prestatario* es varios a uno de *cliente* a *préstamo*—esto es exactamente lo contrario a la interpretación correcta.

Si los dos segmentos de una relación binaria tienen un valor máximo de 1, la relación es uno a uno. Si se hubiese especificado un límite de cardinalidad de 1..* en el segmento entre *cliente* y *prestatario*, se estaría diciendo que cada cliente debe tener, al menos, un préstamo.

6.5 Aspectos del diseño entidad-relación

Los conceptos de conjunto de entidades y de conjunto de relaciones no son precisos, y es posible definir el conjunto de entidades y las relaciones entre ellas de diferentes formas. En este apartado se examinan aspectos básicos del diseño de esquemas de bases de datos E-R. El Apartado 6.7.4 trata el proceso de diseño con más detalle.

6.5.1 Uso de conjuntos de entidades y de atributos

Considérese el conjunto de entidades *empleado* con los atributos *nombre_empleado* y *número_teléfono*. Se puede argumentar que un *teléfono* es una entidad en sí misma con los atributos *número_teléfono* y *ubicación*; la ubicación puede ser la sucursal o el domicilio donde el teléfono está instalado, y se pueden representar los teléfonos móviles (celulares) mediante el valor “móvil”. Si se adopta este punto de vista, hay que redefinir el conjunto de entidades *empleado* como:

- El conjunto de entidades *empleado* con los atributos *id_empleado* y *nombre_empleado*.
- El conjunto de entidades *teléfono* con los atributos *número_teléfono* y *ubicación*.
- La relación *teléfono_empleado*, que denota la asociación entre los empleados y sus teléfonos.

Estas alternativas se muestran en la Figura 6.15.

¿Cuál es, entonces, la diferencia principal entre estas dos definiciones de empleado? Tratar el teléfono como el atributo *número_teléfono* implica que cada empleado tiene exactamente un número de teléfono. Tratar el teléfono como la entidad *teléfono* permite que los empleados tengan varios números de telé-

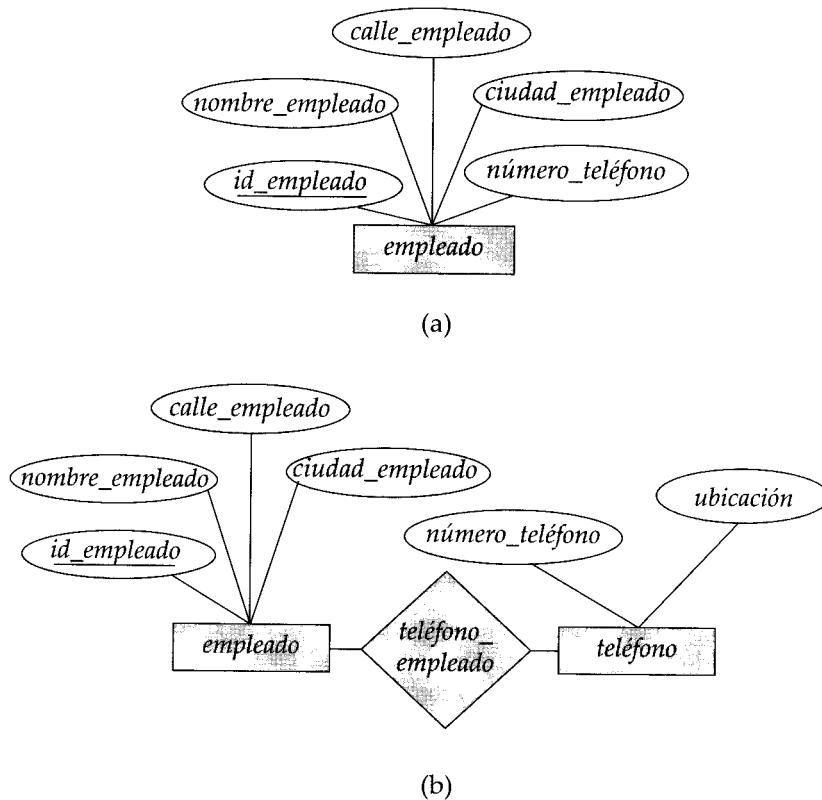


Figura 6.15 Alternativas para *empleado* y *teléfono*.

fono (o ninguno) asociados. Sin embargo, se puede definir en su lugar *número_telefono* como atributo multivalorado para permitir varios teléfonos por empleado.

La diferencia principal es que tratar el teléfono como entidad modela mejor una situación en la que se puede querer guardar información extra sobre el teléfono, como su ubicación, su tipo (móvil, videoteleéfono o fijo) o las personas que lo comparten. Por tanto, tratar el teléfono como entidad es más general que tratarlo como atributo y resulta adecuado cuando la generalidad pueda ser útil.

En cambio, no sería apropiado tratar el atributo *nombre_empleado* como entidad; es difícil argumentar que *nombre_empleado* sea una entidad en sí misma (a diferencia de lo que ocurre con *teléfono*). Así pues, resulta adecuado tener *nombre_empleado* como atributo del conjunto de entidades *empleado*.

Por tanto, se suscitan naturalmente dos preguntas: ¿qué constituye un atributo? y ¿qué constituye un conjunto de entidades? Desafortunadamente no hay respuestas sencillas. Las distinciones dependen principalmente de la estructura de la empresa real que se esté modelando y de la semántica asociada con el atributo en cuestión.

Un error común es usar la clave primaria de un conjunto de entidades como atributo de otro conjunto de entidades en lugar de usar una relación. Por ejemplo, es incorrecto modelar *id_cliente* como atributo de *préstamo*, aunque cada préstamo tenga sólo un cliente. La relación *prestatario* es la forma correcta de representar la conexión entre los préstamos y los clientes, ya que hace explícita su conexión, en lugar de dejarla implícita mediante un atributo.

Otro error relacionado con éste que se comete a veces es escoger los atributos de clave primaria de los conjuntos de entidades relacionados como atributos del conjunto de relaciones. Por ejemplo, *número_préstamo* (el atributo de clave primaria de *préstamo*) e *id_cliente* (la clave primaria de *cliente*) no deben aparecer como atributos de la relación *prestatario*. Esto no es adecuado, ya que los atributos de clave primaria están implícitos en el conjunto de relaciones³.

3. Cuando se crea un esquema de relación a partir del esquema E-R los atributos pueden aparecer en una tabla creada a partir del conjunto de relaciones *prestatario*, como se verá más adelante; no obstante, no deben aparecer en el conjunto de relaciones *prestatario*.

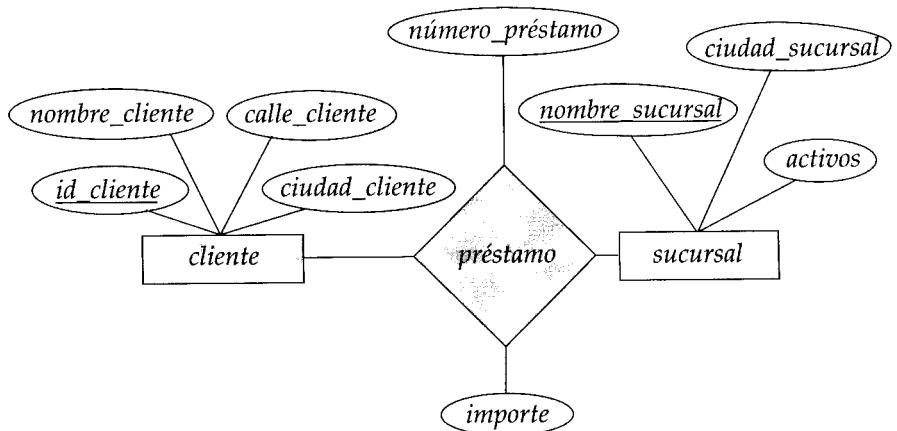


Figura 6.16 préstamo como conjunto de relaciones.

6.5.2 Uso de los conjuntos de entidades y de los conjuntos de relaciones

No siempre está claro si es mejor expresar un objeto mediante un conjunto de entidades o mediante un conjunto de relaciones. En el Apartado 6.2.1 se dio por supuesto que los préstamos bancarios se modelaban como entidades. Una alternativa es modelar los préstamos no como entidades, sino como relaciones entre los clientes y las sucursales, con *número_préstamo* e *importe* como atributos descriptivos, como puede verse en la Figura 6.16. Cada préstamo se representa mediante una relación entre un cliente y una sucursal.

Si cada préstamo se concede exactamente a un cliente y se asocia exactamente con una sucursal, se puede encontrar satisfactorio el diseño en el que cada préstamo se representa como una relación. Sin embargo, con este diseño, no se puede representar convenientemente una situación en la que un préstamo se conceda a varios clientes conjuntamente. Para tratar esta situación se debe definir otra relación para cada prestatario de ese préstamo conjunto. Entonces habrá que replicar los valores de los atributos descriptivos *número_préstamo* e *importe* en cada una de esas relaciones. Cada una de esas relaciones debe, por supuesto, tener el mismo valor para los atributos descriptivos *número_préstamo* e *importe*.

Se plantean dos problemas como resultado de esta réplica: (1) los datos se almacenan varias veces, lo que hace que se desperdicie espacio de almacenamiento, y (2) las actualizaciones pueden dejar los datos en un estado inconsistente, en el que los valores de atributos que se supone que tienen el mismo valor difieran. El problema de cómo evitar esta réplica se trata formalmente mediante la *teoría de la normalización*, que se aborda en el Capítulo 7.

El problema de la réplica de los atributos *número_préstamo* e *importe* no aparece en el diseño original del Apartado 6.4, ya que allí *préstamo* es un conjunto de entidades.

Un criterio para determinar si se debe usar un conjunto de entidades o un conjunto de relaciones puede ser escoger un conjunto de relaciones para describir las acciones que ocurrán entre entidades. Este enfoque también puede ser útil para decidir si ciertos atributos se pueden expresar mejor como relaciones.

6.5.3 Conjuntos de relaciones binarias y n-arias

Las relaciones en las bases de datos suelen ser binarias. Puede que algunas relaciones que no parecen ser binarias se puedan representar mejor mediante varias relaciones binarias. Por ejemplo, se puede crear la relación ternaria *padres*, que relaciona a cada hijo con su padre y con su madre. Sin embargo, esa relación se puede representar mediante dos relaciones binarias, *padre* y *madre*, que relacionan a cada hijo con su padre y con su madre por separado. El uso de las dos relaciones *padre* y *madre* permite el registro de la madre del niño aunque no se conozca la identidad del padre; si se usara en la relación ternaria *padres* se necesitaría un valor nulo. En este caso es preferible usar conjuntos de relaciones binarias.

De hecho, siempre es posible sustituir los conjuntos de relaciones no binarias (*naria*, para $n > 2$) por varios conjuntos de relaciones binarias. Por simplificar, considérense el conjunto de relaciones abstracto ternario ($n = 3$) R y los conjuntos de entidades A , B , y C . Se sustituye el conjunto de relaciones R por un conjunto de entidades E y se crean tres conjuntos de relaciones como se muestra en la Figura 6.17:

- P_A , que relaciona E y A
- P_B , que relaciona E y B
- P_C , que relaciona E y C

Si el conjunto de relaciones R tiene atributos, éstos se asignan al conjunto de entidades E ; además, se crea un atributo de identificación especial para E (ya que se debe poder distinguir las diferentes entidades de cada conjunto de entidades con base en los valores de sus atributos). Para cada relación (a_i, b_i, c_i) del conjunto de relaciones R se crea una nueva entidad e_i del conjunto de entidades E . Luego, en cada uno de los tres nuevos conjuntos de relaciones, se inserta una relación del modo siguiente:

- (e_i, a_i) en R_A
- (e_i, b_i) en R_B
- (e_i, c_i) en R_C

Este proceso se puede generalizar de forma directa a los conjuntos de relaciones n -arias. Por tanto, conceptualmente, se puede restringir el modelo E-R para que sólo incluya conjuntos de relaciones binarias. Sin embargo, esta restricción no siempre es deseable.

- Es posible que sea necesario crear un atributo de identificación para que el conjunto de entidades represente el conjunto de relaciones. Este atributo, junto con los conjuntos de relaciones adicionales necesarios, incrementa la complejidad del diseño y (como se verá en el Apartado 6.9) los requisitos globales de almacenamiento.
- Un conjunto de relaciones n -arias muestra más claramente que varias entidades participan en una sola relación.
- Puede que no haya forma de traducir las restricciones a la relación ternaria en restricciones a las relaciones binarias. Por ejemplo, considérese una restricción que dice que R es varios a uno de A , B a C ; es decir, cada par de entidades de A y de B se asocia, a lo sumo, con una entidad de C . Esta restricción no se puede expresar mediante restricciones de cardinalidad sobre los conjuntos de relaciones R_A , R_B y R_C .

Considérese el conjunto de relaciones *trabaja_en* del Apartado 6.2.2, que relaciona *empleado*, *sucursal* y *trabajo*. No se puede dividir directamente *trabaja_en* en relaciones binarias entre *empleado* y *sucursal* y entre *empleado* y *trabajo*. Si se hiciera, se podría registrar que Santos es director e interventor y que Santos trabaja en Navacerrada y en Centro; sin embargo, no se podría registrar que Santos es director de Navacerrada e interventor de Centro, pero que no es interventor de Navacerrada ni director de Centro.

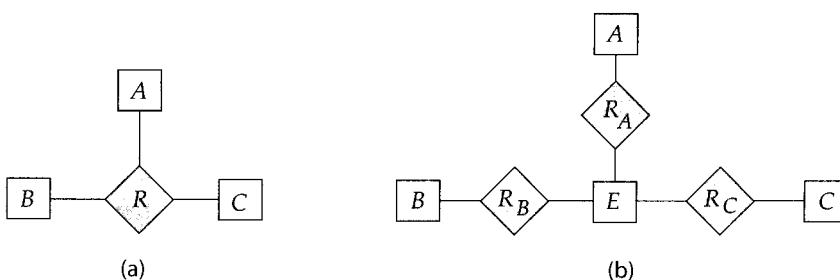


Figura 6.17 Una relación ternaria y tres relaciones binarias.

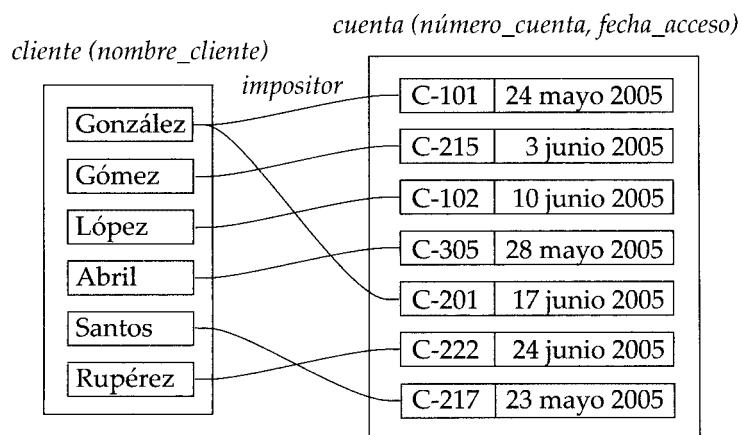


Figura 6.18 *fecha_acceso* como atributo del conjunto de entidades *cuenta*.

El conjunto de relaciones *trabaja_en* se puede dividir en relaciones binarias mediante la creación de nuevos conjuntos de entidades como se ha descrito anteriormente. Sin embargo, no sería muy natural.

6.5.4 Ubicación de los atributos de las relaciones

La razón de cardinalidad de una relación puede afectar a la ubicación de sus atributos. Por tanto, los atributos de los conjuntos de relaciones uno a uno o uno a varios pueden estar asociados con uno de los conjuntos de entidades participantes, en lugar de con el conjunto de relaciones. Por ejemplo, especifiquemos que *impositor* es un conjunto de relaciones uno a varios tal que cada cliente puede tener varias cuentas, pero cada cuenta sólo puede tener un cliente como titular. En este caso, el atributo *fecha_acceso*, que especifica la fecha en que el cliente tuvo acceso a la cuenta por última vez, podría estar asociado con el conjunto de entidades *cuenta*, como se describe en la Figura 6.18; para simplificar la figura sólo se muestran algunos de los atributos de los dos conjuntos de entidades. Dado que cada entidad *cuenta* participa en una relación con un ejemplar de *cliente*, como máximo, hacer esta designación de atributos tendría el mismo significado que colocar *fecha_acceso* en el conjunto de relaciones *impositor*. Los atributos de un conjunto de relaciones uno a varios sólo se pueden recolocar en el conjunto de entidades de la parte “varios” de la relación. Por otra parte, para los conjuntos de entidades uno a uno, los atributos de la relación se pueden asociar con cualquiera de las entidades participantes.

La decisión de diseño sobre la ubicación de los atributos descriptivos en estos casos—como atributo de la relación o de la entidad—debe reflejar las características de la empresa que se modela. El diseñador puede elegir mantener *fecha_acceso* como atributo de *impositor* para expresar explícitamente que se produce un acceso en el punto de interacción entre los conjuntos de entidades *cliente* y *cuenta*.

La elección de la ubicación del atributo es más sencilla para los conjuntos de relaciones varios a varios. Volviendo al ejemplo, especifiquemos el caso, quizá más realista, de que *impositor* sea un conjunto de relaciones varios a varios que expresa que cada cliente puede tener una o más cuentas, y que cada cuenta puede tener a varios clientes como titulares. Si hay que expresar la fecha en que un cliente dado tuvo acceso por última vez a una cuenta concreta, *fecha_acceso* debe ser atributo del conjunto de relaciones *impositor*, en lugar de serlo de cualquiera de las entidades participantes. Si *fecha_acceso* fuese un atributo de *cuenta*, por ejemplo, no se podría determinar qué cliente llevó a cabo el acceso más reciente a una cuenta conjunta. Cuando un atributo se determina mediante la combinación de los conjuntos de entidades participantes, en lugar de por cada entidad por separado, ese atributo debe estar asociado con el conjunto de relaciones varios a varios. La Figura 6.3 muestra la ubicación de *fecha_acceso* como atributo de la relación; de nuevo, para simplificar la figura, sólo se muestran algunos de los atributos de los dos conjuntos de entidades.

6.6 Conjuntos de entidades débiles

Puede que un conjunto de entidades no tenga suficientes atributos para formar una clave primaria. Ese conjunto de entidades se denomina **conjunto de entidades débiles**. Los conjuntos de entidades que tienen una clave primaria se denominan **conjuntos de entidades fuertes**.

Como ilustración, considérese el conjunto de entidades *pago*, que tiene tres atributos: *número_pago*, *fecha_pago* e *importe_pago*. Los números de pago suelen ser números secuenciales, a partir de 1, generados independientemente para cada préstamo. Por tanto, aunque cada entidad *pago* es distinta, los pagos de diferentes préstamos pueden compartir el mismo número de pago. Así, este conjunto de entidades no tiene clave primaria; es un conjunto de entidades débiles.

Para que un conjunto de entidades débiles tenga sentido, debe estar asociado con otro conjunto de entidades, denominado **conjunto de entidades identificadoras o propietarias**. Cada entidad débil debe estar asociada con una entidad identificadora; es decir, se dice que el conjunto de entidades débiles **depende existencialmente** del conjunto de entidades identificadoras. Se dice que el conjunto de entidades identificadoras **posee** el conjunto de entidades débiles al que identifica. La relación que asocia el conjunto de entidades débiles con el conjunto de entidades identificadoras se denomina **relación identificadora**. La relación identificadora es varios a uno del conjunto de entidades débiles al conjunto de entidades identificadoras y la participación del conjunto de entidades débiles en la relación es total.

En nuestro ejemplo, el conjunto de entidades identificador de *pago* es *préstamo*, y la relación *pago_préstamo* que asocia las entidades *pago* con sus correspondientes entidades *préstamo* es la relación identificadora.

Aunque los conjuntos de entidades débiles no tienen clave primaria, hace falta un medio para distinguir entre todas las entidades del conjunto de entidades débiles que dependen de una entidad fuerte concreta. El **discriminante** de un conjunto de entidades débiles es un conjunto de atributos que permite que se haga esta distinción. Por ejemplo, el discriminante del conjunto de entidades débiles *pago* es el atributo *número_pago* ya que, para cada préstamo, el número de pago identifica de forma única cada pago de ese préstamo. El discriminante del conjunto de entidades débiles se denomina *clave parcial* del conjunto de entidades.

La clave primaria de un conjunto de entidades débiles se forma con la clave primaria del conjunto de entidades identificadoras y el discriminante del conjunto de entidades débiles. En el caso del conjunto de entidades *pago*, su clave primaria es {*número_préstamo*, *número_pago*}, donde *número_préstamo* es la clave primaria del conjunto de entidades identificadoras, es decir, *préstamo*, y *número_pago* distingue las entidades *pago* de un mismo préstamo.

El conjunto de entidades identificadoras no debe tener atributos descriptivos, ya que cualquier atributo requerido puede estar asociado con el conjunto de entidades débiles (véase la discusión sobre el traslado de los atributos del conjunto de relaciones a los conjuntos de entidades participantes en el Apartado 6.5.4).

Los conjuntos de entidades débiles pueden participar en otras relaciones aparte de la relación identificadora. Por ejemplo, la entidad *pago* puede participar en una relación con el conjunto de entidades *cuenta*, identificando la cuenta desde la que se ha realizado el pago. Los conjuntos de entidades débiles pueden participar como propietario de una relación identificadora con otro conjunto de entidades débiles. También es posible tener conjuntos de entidades débiles con más de un conjunto de entidades identificadoras. Cada entidad débil se identificaría mediante una combinación de entidades, una de cada conjunto de entidades identificadoras. La clave primaria de la entidad débil consistiría de la unión de las claves primarias de los conjuntos de entidades identificadoras y el discriminante del conjunto de entidades débiles.

En los diagramas E-R los rectángulos con líneas dobles indican conjuntos de entidades débiles, mientras que un rombo con líneas dobles indica la correspondiente relación de identificación. En la Figura 6.19, el conjunto de entidades débiles *pago* depende del conjunto de entidades fuertes *préstamo* mediante el conjunto de relaciones *pago_préstamo*.

La figura también ilustra el uso de líneas dobles para indicar *participación total*—la participación del conjunto de entidades (débiles) *pago* en la relación *pago_préstamo* es total, lo que significa que cada pago debe estar relacionando mediante *pago_préstamo* con alguna cuenta. Finalmente, la flecha de *pago_préstamo*

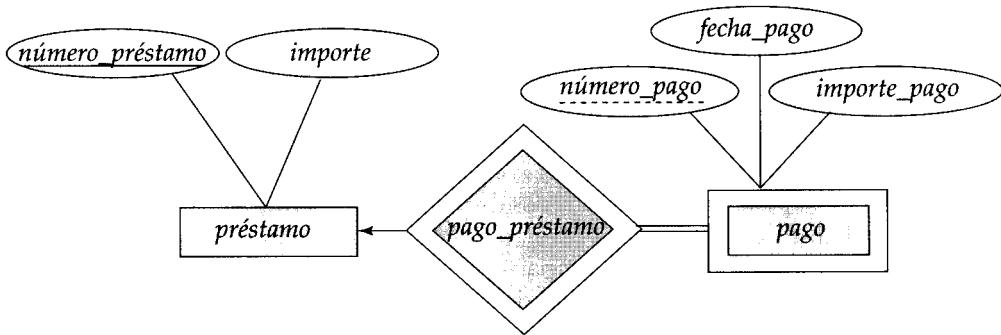


Figura 6.19 Diagrama E-R con un conjunto de entidades débiles.

tamo a préstamo indica que cada pago es para un solo préstamo. Los discriminantes de los conjuntos de entidades débiles también se subrayan, pero con una línea discontinua, en lugar de con una continua.

En algunos casos, puede que el diseñador de la base de datos decida expresar un conjunto de entidades débiles como atributo compuesto multivalorado del conjunto de entidades propietarias. En el ejemplo, esta alternativa exigiría que el conjunto de entidades *préstamo* tuviera el atributo compuesto y multivalorado *pago*, que constara de *número_pago*, *fecha_pago* e *importe_pago*. Los conjuntos de entidades débiles se pueden modelar mejor como atributos si sólo participan en la relación identificadora y tienen pocos atributos. A la inversa, las representaciones de los conjuntos de entidades débiles modelarán mejor las situaciones en las que esos conjuntos participen en otras relaciones aparte de la relación identificadora y tengan muchos atributos.

Como ejemplo adicional de conjunto de entidades que se puede modelar como conjunto de entidades débiles, considérese las ofertas de asignaturas en una universidad. La misma asignatura se puede ofrecer en diferentes cursos y, dentro de cada curso, puede haber varios grupos para la misma asignatura. Por tanto, se puede crear el conjunto de entidades débiles *oferta_asignatura*, que depende existencialmente de *asignatura*; las diferentes ofertas de la misma asignatura se identifican mediante *curso* y *número_grupo*, que forman un discriminante pero no una clave primaria.

6.7 Características del modelo E-R extendido

Aunque los conceptos básicos del modelo E-R pueden modelar la mayor parte de las características de las bases de datos, algunos aspectos de las bases de datos se pueden expresar mejor mediante ciertas extensiones del modelo E-R básico. En este apartado se estudian las características E-R extendidas de especialización, generalización, conjuntos de entidades de nivel superior e inferior, herencia de atributos y agregación.

6.7.1 Especialización

Los conjuntos de entidades pueden incluir subgrupos de entidades que se diferencian de alguna forma de las demás entidades del conjunto. Por ejemplo, un subconjunto de entidades de un conjunto de entidades puede tener atributos que no sean compartidos por todas las entidades del conjunto de entidades. El modelo E-R ofrece un medio de representar estos grupos de entidades diferentes.

Como ejemplo, considérese el conjunto de entidades *persona* con los atributos *id_persona*, *nombre*, *calle* y *ciudad*. Cada persona puede clasificarse además en una de las categorías siguientes:

- *cliente*
- *empleado*

Cada uno de estos tipos de persona se describen mediante un conjunto de atributos que incluye todos los atributos del conjunto de entidades *persona* más otros posibles atributos adicionales. Por ejemplo, las entidades *cliente* se pueden describir además mediante el atributo *calificación_crediticia*, mientras que

las entidades *empleado* se pueden describir además mediante el atributo *sueldo*. El proceso de establecimiento de subgrupos dentro del conjunto de entidades se denomina **especialización**. La especialización de *persona* permite distinguir entre las personas basándonos en si son empleados o clientes: en general, cada persona puede ser empleado, cliente, las dos cosas o ninguna de ellas.

Como ejemplo adicional, supóngase que el banco desea dividir las cuentas en dos categorías: cuentas corrientes y cuentas de ahorro. Las cuentas de ahorro necesitan un saldo mínimo, pero el banco puede establecer diferentes tasas de interés para cada cliente y ofrecer mejores tasas a los clientes preferentes. Las cuentas corrientes tienen una tasa de interés fija, pero permiten los descubiertos; hay que registrar el importe de los descubiertos de las cuentas corrientes. Cada uno de estos tipos de cuenta se describe mediante un conjunto de atributos que incluye todos los atributos del conjunto de entidades *cuenta* más otros atributos adicionales.

El banco puede crear dos especializaciones de *cuenta*, por ejemplo, *cuenta_ahorro* y *cuenta_corriente*. Como ya se ha visto, las entidades *cuenta* se describen mediante los atributos *número_cuenta* y *saldo*. El conjunto de entidades *cuenta_ahorro* tendría todos los atributos de *cuenta* y el atributo adicional *tasa_interés*. El conjunto de entidades *cuenta_corriente* tendría todos los atributos de *cuenta* y el atributo adicional *importe_descubierto*.

La especialización se puede aplicar repetidamente para refinar el esquema de diseño. Por ejemplo, los empleados del banco se pueden clasificar también en alguna de las categorías siguientes:

- *oficial*
- *cajero*
- *secretaria*

Cada uno de estos tipos de empleado se describe mediante un conjunto de atributos que incluye todos los atributos del conjunto de entidades *empleado* y otros adicionales. Por ejemplo, las entidades *oficial* se pueden describir además por el atributo *número_despacho*, las entidades *cajero* por los atributos *número_caja* y *horas_semana*, y las entidades *secretaria* por el atributo *horas_semana*. Además, las entidades *secretaria* pueden participar en la relación *secretaria_de*, que identifica a los empleados a los que ayuda cada secretaria.

Cada conjunto de entidades se puede especializar en más de una característica distintiva. En este ejemplo, la característica distintiva entre las entidades *empleado* es el trabajo que desempeña cada empleado. Otra especialización coexistente se puede basar en si cada persona es un trabajador temporal o fijo, lo que da lugar a los conjuntos de entidades *empleado_temporal* y *empleado_fijo*. Cuando se forma más de una especialización en un conjunto de entidades, cada entidad concreta puede pertenecer a varias especializaciones. Por ejemplo, un empleado dado puede ser un empleado temporal y una secretaria.

En términos de los diagramas E-R, la especialización se representa mediante un componente triangular etiquetado ES, como muestra la Figura 6.20. La etiqueta ES representa, por ejemplo, que cada cliente “es” una persona. La relación ES también se puede denominar relación **superclase-subclase**. Los conjuntos de entidades de nivel superior e inferior se representan como conjuntos de entidades regulares—es decir, como rectángulos que contienen el nombre del conjunto de entidades.

6.7.2 Generalización

El refinamiento a partir del conjunto de entidades inicial en sucesivos niveles de subgrupos de entidades representa un proceso de diseño **descendente** en el que las distinciones se hacen explícitas. El proceso de diseño también puede proceder de forma **ascendente**, en la que varios conjuntos de entidades se sintetizan en un conjunto de entidades de nivel superior basado en características comunes. El diseñador de la base de datos puede haber identificado primero el conjunto de entidades *cliente* con los atributos *id_cliente*, *nombre_cliente*, *calle_cliente*, *ciudad_cliente* y *calificación_crediticia*, y el conjunto de entidades *empleado* con los atributos *id_empleado*, *nombre_empleado*, *calle_empleado*, *ciudad_empleado* y *sueldo_empleado*.

Existen analogías entre el conjunto de entidades *cliente* y el conjunto de entidades *empleado* en el sentido de que tienen varios atributos que, conceptualmente, son iguales en los dos conjuntos de entidades: los atributos para el identificador, el nombre, la calle y la ciudad. Esta similitud se puede expresar mediante la **generalización**, que es una relación de contención que existe entre el conjunto de entidades de

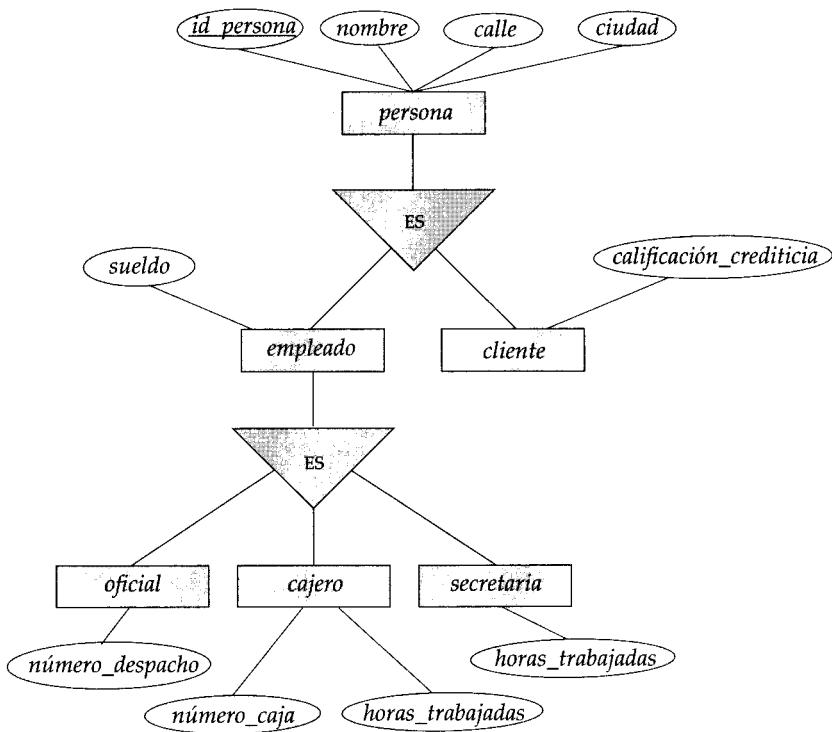


Figura 6.20 Especialización y generalización.

nivel superior y uno o varios conjuntos de entidades de *nivel inferior*. En este ejemplo, *persona* es el conjunto de entidades de nivel superior y *cliente* y *empleado* son conjuntos de entidades de nivel inferior. En este caso, los atributos que son conceptualmente iguales tienen nombres diferentes en los dos conjuntos de entidades de nivel inferior. Para crear generalizaciones los atributos deben tener un nombre común y representarse mediante la entidad de nivel superior *persona*. Se pueden usar los nombres de atributos *id_persona*, *nombre*, *calle* y *ciudad*, como se vio en el ejemplo del Apartado 6.7.1.

Los conjuntos de entidades de nivel superior e inferior también se pueden denominar con los términos **superclase** y **subclase**, respectivamente. El conjunto de entidades *persona* es la superclase de las subclases *cliente* y *empleado*.

A todos los efectos prácticos, la generalización es una inversión simple de la especialización. Se aplicarán ambos procesos, combinados, en el transcurso del diseño del esquema E-R de una empresa. En términos del propio diagrama E-R no se distingue entre especialización y generalización. Los niveles nuevos de representación de las entidades se distinguen (especialización) o sintetizan (generalización) cuando el esquema de diseño llega a expresar completamente la aplicación de la base de datos y los requisitos del usuario de la base de datos. Las diferencias entre los dos enfoques se pueden caracterizar mediante su punto de partida y su objetivo global.

La especialización parte de un único conjunto de entidades; destaca las diferencias entre las entidades del conjunto mediante la creación de diferentes conjuntos de entidades de nivel inferior. Esos conjuntos de entidades de nivel inferior pueden tener atributos o participar en relaciones que no se aplican a todas las entidades del conjunto de entidades de nivel inferior. Realmente, la razón de que el diseñador aplique la especialización es poder representar esas características distintivas. Si *cliente* y *empleado* no tuvieran atributos que no tuvieran las entidades *persona* ni participaran en relaciones diferentes de las relaciones en las que participan las entidades *persona*, no habría necesidad de especializar el conjunto de entidades *persona*.

La generalización parte del reconocimiento de que varios conjuntos de entidades comparten algunas características comunes (es decir, se describen mediante los mismos atributos y participan en los mismos conjuntos de relaciones). Con base en esas similitudes, la generalización sintetiza esos conjuntos de entidades en un solo conjunto de nivel superior. La generalización se usa para destacar las similitudes

entre los conjuntos de entidades de nivel inferior y para ocultar las diferencias; también permite una economía de representación, ya que no se repiten los atributos compartidos.

6.7.3 Herencia de los atributos

Una propiedad crucial de las entidades de nivel superior e inferior creadas mediante la especialización y la generalización es la **herencia de los atributos**. Se dice que los atributos de los conjuntos de entidades de nivel superior son **heredados** por los conjuntos de entidades de nivel inferior. Por ejemplo, *cliente* y *empleado* heredan los atributos de *persona*. Así, *cliente* se describe mediante sus atributos *nombre*, *calle* y *ciudad* y, adicionalmente, por el atributo *id_cliente*; *empleado* se describe mediante sus atributos *nombre*, *calle* y *ciudad* y, adicionalmente, por los atributos *id_empleado* y *sueldo*.

Los conjuntos de entidades de nivel inferior (o subclases) también heredan la participación en los conjuntos de relaciones en los que participa su entidad de nivel superior (o superclase). Los conjuntos de entidades *oficial*, *cajero* y *secretaria* pueden participar en el conjunto de relaciones *trabaja_para*, ya que la superclase *empleado* participa en la relación *trabaja_para*. La herencia de los atributos se aplica a todas las capas de conjuntos de entidades de nivel inferior. Los conjuntos de entidades anteriores pueden participar en cualquier relación en la que participe el conjunto de entidades *persona*.

Tanto si se llega a una porción dada del modelo E-R mediante la especialización como si se hace mediante la generalización, el resultado es básicamente el mismo:

- Un conjunto de entidades de nivel superior con los atributos y las relaciones que se aplican a todos sus conjuntos de entidades de nivel inferior.
- Conjuntos de entidades de nivel inferior con características distintivas que sólo se aplican en un conjunto dado de entidades de nivel inferior.

En lo que sigue, aunque a menudo sólo se haga referencia a la generalización, las propiedades que se estudian corresponden completamente a ambos procesos.

La Figura 6.20 describe una **jerarquía** de conjuntos de entidades. En la figura, *empleado* es un conjunto de entidades de nivel inferior de *persona* y un conjunto de entidades de nivel superior de los conjuntos de entidades *oficial*, *cajero* y *secretaria*. En las jerarquías un conjunto de entidades dado sólo puede estar implicado como conjunto de entidades de nivel inferior en una relación ES; es decir, los conjuntos de entidades de este diagrama sólo tienen **herencia única**. Si un conjunto de entidades es un conjunto de entidades de nivel inferior en más de una relación ES, el conjunto de entidades tiene **herencia múltiple**, y la estructura resultante se denomina *retículo*.

6.7.4 Restricciones a las generalizaciones

Para modelar una empresa con más precisión, el diseñador de la base de datos puede decidir imponer ciertas restricciones sobre una generalización concreta. Un tipo de restricción implica la determinación de las entidades que pueden formar parte de un conjunto de entidades de nivel inferior dado. Esta pertenencia puede ser una de las siguientes:

- **Definida por la condición.** En los conjuntos de entidades de nivel inferior definidos por la condición, la pertenencia se evalúa en función del cumplimiento de una condición o predicado explícito por la entidad. Por ejemplo, supóngase que el conjunto de entidades de nivel superior *cuenta* tiene el atributo *tipo_cuenta*. Todas las entidades *cuenta* se evalúan según el atributo *tipo_cuenta* que las define. Sólo las entidades que satisfacen la condición *tipo_cuenta* = “cuenta de ahorro” pueden pertenecer al conjunto de entidades de nivel inferior *cuenta_ahorro*. Todas las entidades que satisfacen la condición *tipo_cuenta* = “cuenta corriente” se incluyen en *cuenta_corriente*. Dado que todas las entidades de nivel inferior se evalúan en función del mismo atributo (en este caso, *tipo_cuenta*), se dice que este tipo de generalización está **definida por el atributo**.
- **Definida por el usuario.** Los conjuntos de entidades de nivel inferior definidos por el usuario no están restringidos por una condición de pertenencia; más bien, el usuario de la base de datos asigna las entidades a un conjunto de entidades dado. Por ejemplo, supóngase que, después de tres meses de trabajo, los empleados del banco se asignan a uno de los cuatro grupos de trabajo.

En consecuencia, los grupos se representan como cuatro conjuntos de entidades de nivel inferior del conjunto de entidades de nivel superior *empleado*. No se asigna cada empleado a una entidad grupo concreta automáticamente de acuerdo con una condición explícita que lo defina. En vez de eso, la asignación al grupo la lleva a cabo el usuario que toma persona a persona. La asignación se implementa mediante una operación que añade cada entidad a un conjunto de entidades.

Un segundo tipo de restricciones tiene relación con la pertenencia de las entidades a más de un conjunto de entidades de nivel inferior de la generalización. Los conjuntos de entidades de nivel inferior pueden ser de uno de los tipos siguientes:

- **Disjuntos.** La *restricción sobre la condición de disjunción* exige que cada entidad no pertenezca a más de un conjunto de entidades de nivel inferior. En el ejemplo, cada entidad *cuenta* sólo puede cumplir una condición del atributo *tipo_cuenta*; cada entidad puede ser una cuenta de ahorro o una cuenta corriente, pero no ambas cosas a la vez.
- **Solapados.** En las *generalizaciones solapadas* la misma entidad puede pertenecer a más de un conjunto de entidades de nivel inferior de la generalización. Como ilustración, considérese el ejemplo del grupo de trabajo de empleados y supóngase que algunos directores participan en más de un grupo de trabajo. Cada empleado, por tanto, puede aparecer en más de uno de los conjuntos de entidades grupo que son conjuntos de entidades de nivel inferior de *empleado*. Por tanto, la generalización es solapada.

Como ejemplo adicional, supóngase que la generalización aplicada a los conjuntos de entidades *cliente* y *empleado* conduce a un conjunto de entidades de nivel superior *persona*. La generalización es solapada si los empleados también pueden ser clientes.

El solapamiento de las entidades de nivel inferior es el caso predeterminado; la restricción sobre la condición de disjunción se debe imponer explícitamente a la generalización (o especialización). La restricción sobre condición de disjunción se puede denotar en los diagramas E-R añadiendo la palabra *disjunta* junto al símbolo del triángulo.

Una última restricción, la **restricción de completitud** sobre una generalización o especialización, especifica si una entidad del conjunto de entidades de nivel superior debe pertenecer, al menos, a uno de los conjuntos de entidades de nivel inferior de la generalización o especialización. Esta restricción puede ser de uno de los tipos siguientes:

- **Generalización o especialización total.** Cada entidad de nivel superior debe pertenecer a un conjunto de entidades de nivel inferior.
- **Generalización o especialización parcial.** Puede que alguna entidad de nivel superior no pertenezca a ningún conjunto de entidades de nivel inferior.

La generalización parcial es la predeterminada. Se puede especificar la generalización total en los diagrama E-R usando una línea doble para conectar el rectángulo que representa el conjunto de entidades de nivel superior con el símbolo del triángulo. (Esta notación es parecida a la usada para la participación total en una relación).

La generalización de *cuenta* es total: todas las entidades *cuenta* deben ser cuentas de ahorro o cuentas corrientes. Como el conjunto de entidades de nivel superior al que se llega mediante la generalización suele estar compuesto únicamente de entidades de los conjuntos de entidades de nivel inferior, la restricción de completitud para los conjuntos de entidades de nivel superior generalizados suele ser total. Cuando la restricción es parcial, las entidades de nivel superior no están limitadas a aparecer en los conjuntos de entidades de nivel inferior. Los conjuntos de entidades grupo de trabajo ilustran una especialización parcial. Como los empleados sólo se asignan a cada grupo después de llevar tres meses en el trabajo, puede que algunas entidades *empleado* no pertenezcan a ninguno de los conjuntos de entidades grupo de nivel inferior.

Los conjuntos de entidades equipo se pueden caracterizar mejor como especialización de *empleado* parcial y solapada. La generalización de *cuenta_corriente* y *cuenta_ahorro* en *cuenta* es una generalización total y disjunta. Las restricciones de completitud y sobre la condición de disjunción, sin embargo, no

dependen una de la otra. Las características de las restricciones también pueden ser parcial—disjunta y total—solapada.

Es evidente que algunos requisitos de inserción y de borrado son consecuencia de las restricciones que se aplican a una generalización o especialización dada. Por ejemplo, cuando se impone una restricción de completitud total, las entidades insertadas en un conjunto de entidades de nivel superior se deben insertar, al menos, en uno de los conjuntos de entidades de nivel inferior. Con una restricción de definición por condición, todas las entidades de nivel superior que cumplen la condición se deben insertar en ese conjunto de entidades de nivel inferior. Finalmente, las entidades que se borren de los conjuntos de entidades de nivel superior, se deben borrar también de todos los conjuntos de entidades de nivel inferior asociados a los que pertenezcan.

6.7.5 Agregación

Una limitación del modelo E-R es que no es posible expresar relaciones entre las relaciones. Para ilustrar la necesidad de estos constructores, considérese la relación ternaria *trabaja_en*, que se ya se ha visto anteriormente, entre *empleado*, *sucursal* y *trabajo* (véase la Figura 6.12). Supóngase ahora que se desea registrar el director responsable de las tareas realizadas por cada empleado de cada sucursal; es decir, se desea registrar al director responsable de las combinaciones (*empleado*, *sucursal*, *trabajo*). Supóngase que existe un conjunto de entidades *director*.

Una alternativa para representar esta relación es crear una relación cuaternaria *dirige* entre *empleado*, *sucursal*, *trabajo* y *director* (se necesita una relación cuaternaria—una relación binaria entre *director* y *empleado* no permitiría representar las combinaciones (*sucursal*, *trabajo*) de cada empleado que son responsabilidad de cada director). Mediante los constructores de modelado básicos del modelo E-R se obtiene el diagrama E-R de la Figura 6.21 (por simplificar se han omitido los atributos de los conjuntos de entidades).

Parece que los conjuntos de relaciones *trabaja_en* y *dirige* se pueden combinar en un solo conjunto de relaciones. No obstante, no se deben combinar en una sola relación, ya que puede que algunas combinaciones *empleado*, *sucursal*, *trabajo* no tengan director.

No obstante, hay información redundante en la figura obtenida, ya que cada combinación *empleado*, *sucursal*, *trabajo* de *dirige* también está en *trabaja_en*. Si el director fuese un valor en lugar de una entidad *director*, se podría hacer que *director* fuese un atributo multivalorado de la relación *trabaja_en*. Pero eso dificulta (tanto lógicamente como en coste de ejecución) encontrar, por ejemplo, las tripletas *empleado*-*sucursal*-*trabajo* de las que es responsable cada director. Como el director es una entidad *director*, esta alternativa queda descartada en cualquier caso.

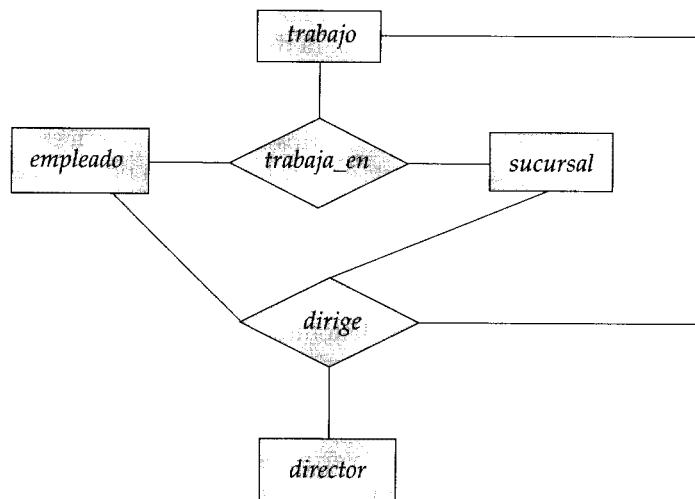


Figura 6.21 Diagrama E-R con relaciones redundantes.

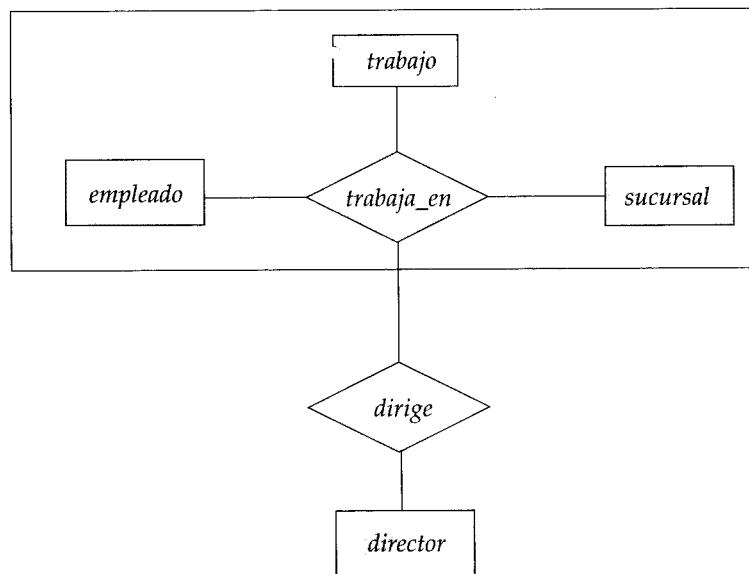


Figura 6.22 Diagrama E-R con agregación.

La mejor forma de modelar una situación como la descrita es usar la agregación. La **agregación** es una abstracción a través de la cual las relaciones se tratan como entidades de nivel superior. Así, para este ejemplo, se considera el conjunto de relaciones *trabaja_en* (que relaciona los conjuntos de entidades *empleado*, *sucursal* y *trabajo*) como el conjunto de entidades de nivel superior denominado *trabaja_en*. Ese conjunto de entidades se trata de la misma forma que cualquier otro conjunto de entidades. Se puede crear entonces la relación binaria *dirige* entre *trabaja_en* y *director* para representar al responsable de cada tarea. La Figura 6.22 muestra una notación para la agregación que se usa habitualmente para representar esta situación.

6.7.6 Notaciones E-R alternativas

La Figura 6.23 resume el conjunto de símbolos que se ha usado en los diagramas E-R. No hay una norma universal para la notación de los diagramas E-R, y cada libro y cada programa informático de diagramas E-R usa notaciones diferentes.

La Figura 6.24 indica alguna de las notaciones alternativas que más se usan. Los conjuntos de entidades se pueden representar como rectángulos con el nombre por fuera, y los atributos relacionados unos debajo de los otros dentro del rectángulo. Los atributos clave primaria se indican relacionándolos en la parte superior, con una línea que los separa de los demás atributos.

Las restricciones de cardinalidad se pueden indicar de varias formas, como se muestra en la Figura 6.24. Las etiquetas * y 1 en los segmentos que salen de las relaciones se usan a menudo para denotar relaciones varios a varios, uno a uno y varios a uno, como muestra la figura. El caso de uno a varios es simétrico con el de varios a uno y no se muestra. En otra notación alternativa de la figura los conjuntos de relaciones se representan mediante líneas entre los conjuntos de entidades, sin rombos; por tanto, sólo se podrán modelar relaciones binarias. Las restricciones de cardinalidad en esta notación se muestran mediante la notación “pata de gallo”, como en la figura.

Desafortunadamente no existe una notación E-R normalizada. La notación que se usa en este libro, con rectángulos, rombos y elipses se denomina notación de Chen, y la usó Chen en el artículo que introdujo el concepto de modelado E-R. El Instituto Nacional de EEUU para Normalización y Tecnología (U.S. National Institute for Standards and Technology) definió una norma denominada IDEF1X en 1993, que usa la notación de pata de gallo. IDEF1X también incluye gran variedad de notaciones diferentes que no se han mostrado, incluidas las barras verticales en los segmentos de las relaciones para indicar participación total y los círculos vacíos para denotar participación parcial. Hay gran variedad de herramientas para la construcción de diagramas E-R, cada una de las cuales tiene sus propias variantes en cuanto a la notación. Véanse las referencias en las notas bibliográficas para obtener más información.

6.8 Diseño de una base de datos para un banco

Ahora se centrará la atención en los requisitos de diseño de la base de datos de una entidad bancaria con más detalle y se desarrollará un diseño más realista, aunque también más complicado, de lo que se ha visto en los ejemplos anteriores. No obstante, no se intentará modelar cada aspecto del diseño de la base de datos para el banco; se considerarán sólo unos cuantos aspectos para ilustrar el proceso de diseño de bases de datos.

Se aplicarán las dos fases iniciales del diseño de bases de datos, es decir, la recopilación de los requisitos de datos y el diseño del esquema conceptual, al ejemplo de entidad bancaria. Se usará el modelo de datos E-R para traducir los requisitos de los usuarios a un esquema de diseño conceptual que se representará como diagrama E-R.

Finalmente, el resultado del proceso de diseño E-R será el esquema de una base de datos relacional. En el Apartado 6.9 se considerará el proceso de generación del diseño relacional a partir de un diseño E-R.

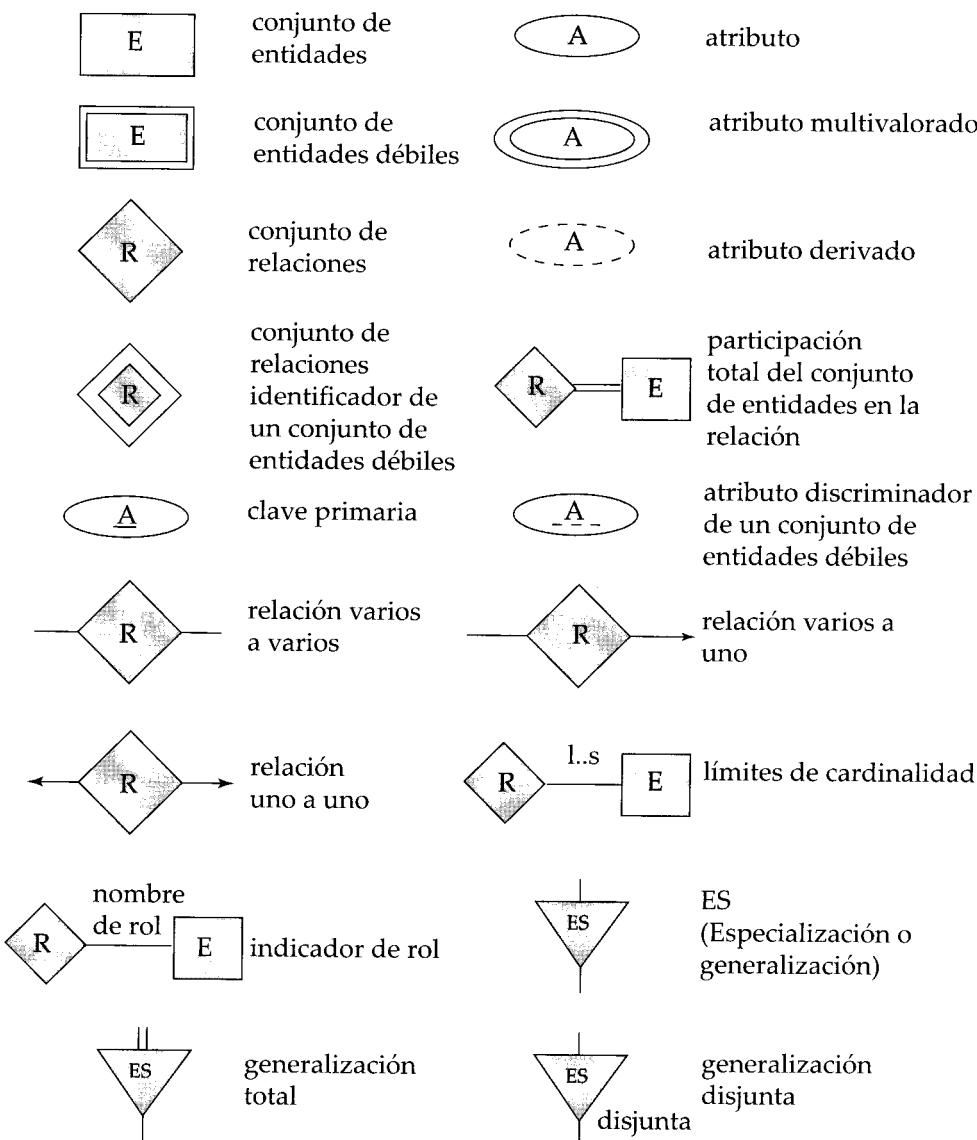
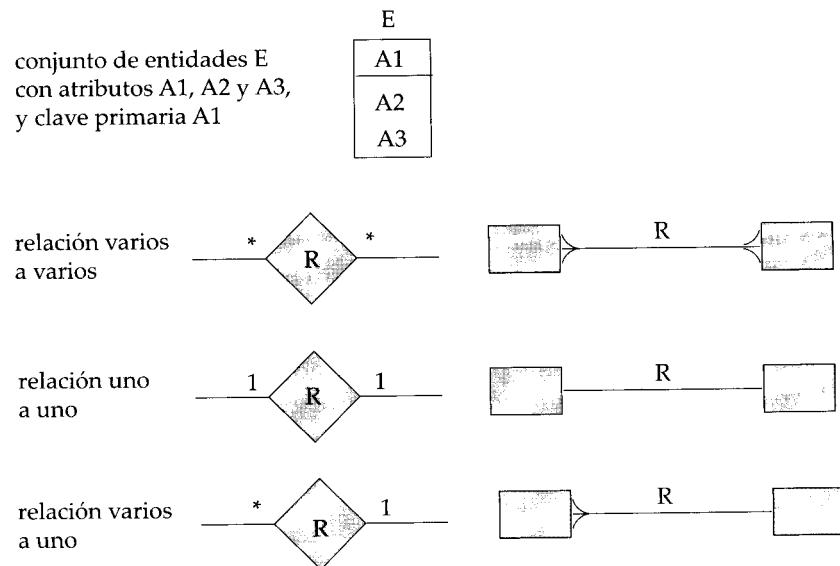


Figura 6.23 Símbolos usados en la notación E-R.

**Figura 6.24** Notaciones E-R alternativas.

Antes de comenzar con el diseño de la base de datos de la entidad bancaria se describirán brevemente las alternativas de diseño E-R entre las que pueden escoger los diseñadores de bases de datos.

6.8.1 Alternativas de diseño E-R

El modelo de datos E-R permite gran flexibilidad en el diseño de los esquemas de las bases de datos para modelar una empresa dada. A continuación se sugiere la manera en que el diseñador de bases de datos puede escoger entre una amplia gama de alternativas. Entre las decisiones que debe tomar el diseñador están:

- Si usar atributos o conjuntos de entidades para representar los objetos (se ha estudiado en el Apartado 6.5.1).
- Si los conceptos del mundo real se expresan mejor mediante conjuntos de entidades o mediante conjuntos de relaciones (Apartado 6.5.2).
- Si usar relaciones ternarias o pares de relaciones binarias (Apartado 6.5.3).
- Si usar conjuntos de entidades fuertes o débiles (Apartado 6.6); cada conjunto de entidades fuertes y sus conjuntos de entidades débiles se puede considerar como un solo “objeto” de la base de datos, ya que las entidades débiles dependen existencialmente de la entidad fuerte.
- Si es adecuado usar la generalización (Apartado 6.7.2); la generalización, o la jerarquía de relaciones ES, contribuye a la modularidad al permitir que los atributos comunes de entidades parecidas se representen en un solo sitio del diagrama E-R.
- Si es adecuado usar la agregación (Apartado 6.7.5); la agregación agrupa parte del diagrama E-R en un solo conjunto de entidades, lo que permite tratar el conjunto de entidades agregadas como una sola unidad sin necesidad de preocuparse por los detalles de su estructura interna.

Se puede ver que los diseñadores de bases de datos necesitan tener una buena comprensión de la empresa que se modela para poder adoptar las diferentes decisiones de diseño necesarias.

6.8.2 Requisitos de datos de la base de datos bancaria

La especificación inicial de los requisitos de los usuarios se puede basar en entrevistas con los usuarios de la base de datos y en el análisis propio del diseñador de la empresa. La descripción que surge de esta

fase de diseño sirve de base para concretar la estructura conceptual de la base de datos. La siguiente lista describe las principales características de la entidad bancaria.

- El banco está organizado en sucursales. Cada sucursal está ubicada en una ciudad concreta y se identifica con un nombre único. El banco supervisa los activos de cada sucursal.
- Los clientes del banco se identifican mediante su valor de *id_cliente*. El banco almacena cada nombre de cliente, y la calle y la ciudad donde vive cada cliente. Los clientes pueden tener cuentas y pueden solicitar préstamos. Cada cliente puede estar asociado con un empleado del banco concreto, que puede actuar como responsable de préstamos o como asesor personal de ese cliente.
- Los empleados del banco se identifican mediante su valor de *id_empleado*. La administración del banco almacena el nombre y el número de teléfono de cada empleado, el nombre de los subordinados de cada empleado, y el número *id_empleado* del jefe de cada empleado. El banco también mantiene un registro de la fecha de incorporación a la empresa del empleado y, por tanto, de su antigüedad.
- El banco ofrece dos tipos de cuentas: cuentas de ahorro y cuentas corrientes. Las cuentas pueden tener como titular a más un cliente, y cada cliente puede tener más de una cuenta. Cada cuenta tiene asignado un número de cuenta único. El banco mantiene un registro del saldo de cada cuenta y de la fecha más reciente en que cada titular de la cuenta tuvo acceso a esa cuenta. Además, cada cuenta de ahorro tiene un tipo de interés y para cada cuenta corriente se registran los descubiertos generados.
- Cada préstamo se genera en una sucursal concreta y pueden solicitarlo uno o más clientes. Cada préstamo se identifica mediante un número de préstamo único. Para cada préstamo el banco mantiene un registro del importe del préstamo y de los pagos realizados y pendientes. Aunque los números de los pagos del préstamo no identifican de forma única cada pago entre los de todos los préstamos del banco, el número de pago sí que identifica cada pago de un préstamo concreto. De cada pago se registran la fecha y el importe.

En las entidades bancarias reales, el banco realiza un seguimiento de los abonos y cargos realizados en las cuentas de ahorros y en las cuentas corrientes, igual que mantiene un registro de los pagos de los préstamos. Debido a que los requisitos del modelo para ese seguimiento son parecidos, y con objeto de mantener las aplicaciones de ejemplo con un tamaño reducido, en este modelo no se realiza el seguimiento de esos abonos y cargos.

6.8.3 Conjuntos de entidades de la base de datos bancaria

La especificación de los requisitos de datos sirve como punto de partida para la construcción del esquema conceptual de la base de datos. A partir de la especificación que aparece en el Apartado 6.8.2 se comienzan a identificar los conjuntos de entidades y sus atributos:

- El conjunto de entidades *sucursal*, con los atributos *nombre_sucursal*, *ciudad_sucursal* y *activos*.
- El conjunto de entidades *cliente*, con los atributos *id_cliente*, *nombre_cliente*, *calle_cliente* y *ciudad_cliente*. Un posible atributo adicional es *nombre_asesor*.
- El conjunto de entidades *empleado*, con los atributos *id_empleado*, *nombre_empleado*, *número_teléfono*, *suelo* y *jefe*. Algunas características descriptivas adicionales son el atributo multivalorado *nombre_subordinado*, el atributo básico *fecha_contratación* y el atributo derivado *antigüedad*.
- Dos conjuntos de entidades cuenta—*cuenta_ahorro* y *cuenta_corriente*—con los atributos comunes *número_cuenta* y *saldo*; además, *cuenta_ahorro* tiene el atributo *tipo_interés* y *cuenta_corriente* tiene el atributo *descubierto*.
- El conjunto de entidades *préstamo*, con los atributos *número_préstamo*, *importe* y *sucursal_origen*.
- El conjunto de entidades débiles *pago_préstamo*, con los atributos *número_pago*, *fecha_pago* e *importe_pago*.

6.8.4 Conjuntos de relaciones de la base de datos bancaria

Volviendo ahora al esquema de diseño rudimentario del Apartado 6.8.3 se pueden especificar los conjuntos de relaciones y correspondencias de cardinalidades siguientes. En el proceso también se perfeccionan algunas de las decisiones tomadas anteriormente en relación con los atributos de los conjuntos de entidades.

- *prestatario*, un conjunto de relaciones varios a varios entre *cliente* y *préstamo*.
- *sucursal_préstamo*, un conjunto de relaciones varios a uno que indica la sucursal en que se ha originado un préstamo. Obsérvese que este conjunto de relaciones sustituye al atributo *sucursal_origen* del conjunto de entidades *préstamo*.
- *pago_préstamo*, un conjunto de relaciones uno a varios de *préstamo* a *pago*, que documenta que se ha realizado un pago de un préstamo.
- *impositor*, con el atributo de relación *fecha_acceso*, un conjunto de relaciones varios a varios entre *cliente* y *cuenta*, que indica que un cliente posee una cuenta.
- *asesor_personal*, con el atributo de relación *tipo*, un conjunto de relaciones varios a uno que expresa que un cliente puede ser asesorado por un empleado del banco, y que un empleado del banco puede asesorar a uno o más clientes. Obsérvese que este conjunto de relaciones ha sustituido al atributo *nombre_asesor* del conjunto de entidades *cliente*.
- *trabaja_para*, un conjunto de relaciones entre entidades *empleado* con los indicadores de rol *jefe* y *trabajador*; la correspondencia de cardinalidades expresa que cada empleado trabaja para un único jefe, y que cada jefe supervisa a uno o más empleados. Obsérvese que este conjunto de relaciones ha sustituido al atributo *jefe* de *empleado*.

6.8.5 Diagrama E-R de la base de datos bancaria

Conforme a lo estudiado en el Apartado 6.8.4 se presenta ahora el diagrama E-R completo del ejemplo de la entidad bancaria. La Figura 6.25 muestra la representación completa de un modelo conceptual del banco, expresada en términos de los conceptos E-R. El diagrama incluye los conjuntos de entidades, los atributos, los conjuntos de relaciones y las correspondencias de cardinalidades a los que se ha llegado mediante el proceso de diseño de los Apartados 6.8.2 y 6.8.3 y que se han perfeccionado en el Apartado 6.8.4.

El diagrama E-R de esta visión simplificada de una entidad bancaria ya es bastante complejo. Los diagramas E-R de las empresas reales no se pueden dibujar en una sola página y hay que dividirlos en varias partes. Puede hacer falta que las entidades aparezcan varias veces en diferentes partes del diagrama. Los atributos de cada entidad se muestran en una sola aparición de la entidad (preferiblemente la primera) y todas las demás apariciones de la entidad se muestran sin atributos.

6.9 Reducción a esquemas relacionales

Las bases de datos que se ajustan a un esquema de bases de datos E-R se pueden representar mediante conjuntos de esquemas de relación. Para cada conjunto de entidades y para cada conjunto de relaciones de la base de datos hay un solo esquema de relación a la que se asigna el nombre del conjunto de entidades o del conjunto de relaciones correspondiente.

Tanto el modelo E-R de bases de datos como el relacional son representaciones abstractas y lógicas de empresas del mundo real. Como los dos modelos usan principios de diseño parecidos, los diseños E-R se pueden convertir en diseños relacionales.

En este apartado se describe la manera de representar los esquemas E-R mediante esquemas de relación y el modo de asignar las restricciones que surgen del modelo E-R a restricciones de los esquemas de relación.

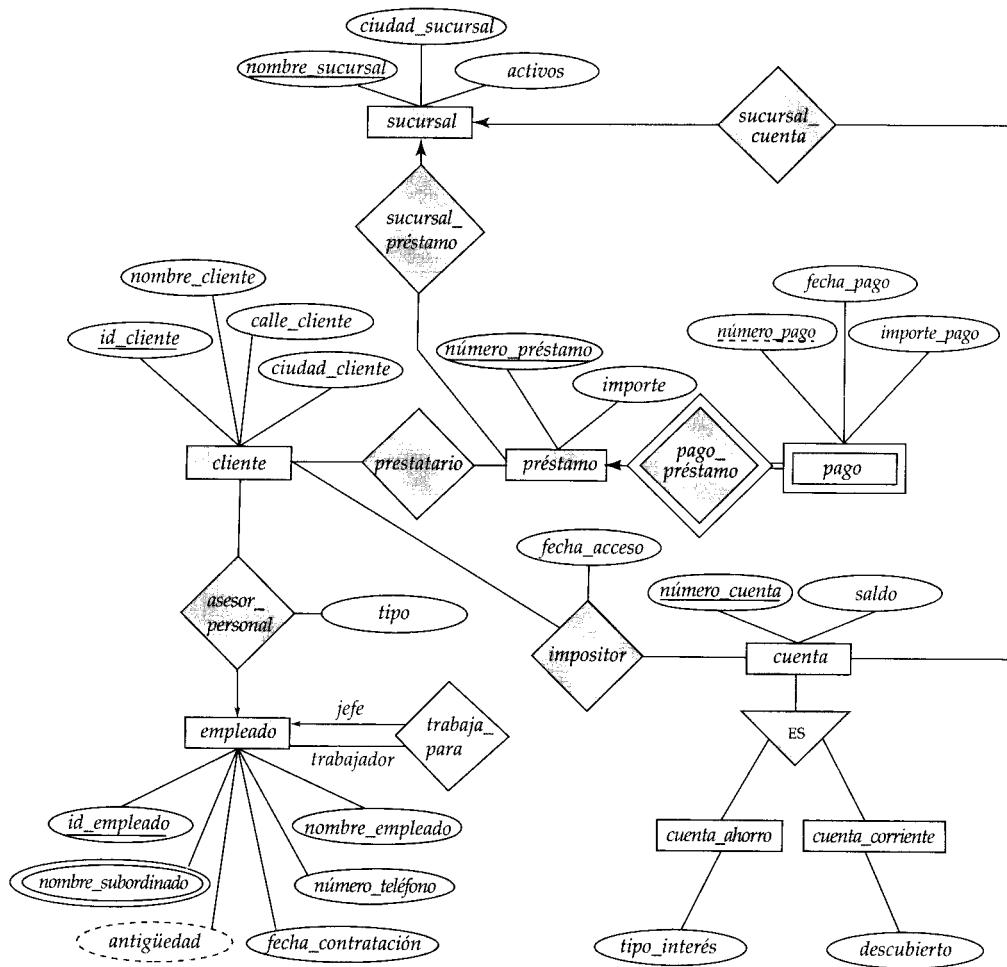


Figura 6.25 Diagrama E-R de una entidad bancaria.

6.9.1 Representación de los conjuntos de entidades fuertes

Sea E un conjunto de entidades fuertes con los atributos descriptivos a_1, a_2, \dots, a_n . Esta entidad se representa mediante un esquema denominado E con n atributos distintos. Cada tupla de las relaciones de este esquema corresponde a una entidad del conjunto de entidades E (más adelante, en el Apartado 6.9.4, se describe la manera de tratar los atributos compuestos y los multivalorados).

Para los esquemas derivados de los conjuntos de entidades fuertes la clave primaria del conjunto de entidades sirve de clave primaria de los esquemas resultantes. Esto se deduce directamente del hecho de que cada tupla corresponde a una entidad concreta del conjunto de entidades.

Como ilustración, considérese el conjunto de entidades *préstamo* del diagrama E-R de la Figura 6.7. Este conjunto de entidades tiene dos atributos: *número_préstamo* e *importe*. Este conjunto de entidades se representa mediante un esquema denominado *préstamo*, con dos atributos:

préstamo = (*número_prestamo*, *importe*)

Obsérvese que, como *número_préstamo* es la clave primaria del conjunto de entidades, también es la clave primaria del esquema de la relación.

En la Figura 6.26 se muestra una relación de este esquema. La tupla

(P-17, 1.000)

<i>número_préstamo</i>	<i>importe</i>
P-11	900
P-14	1.500
P-15	1.500
P-16	1.300
P-17	1.000
P-23	2.000
P-93	500

Figura 6.26 La tabla *préstamo*

significa que el número de préstamo P-17 tiene un importe de 1.000 €. Se pueden añadir entidades nuevas a la base de datos insertando tuplas en las relaciones correspondientes. También se pueden borrar o modificar entidades modificando las tuplas correspondientes.

6.9.2 Representación de los conjuntos de entidades débiles

Sea *A* un conjunto de entidades débiles con los atributos a_1, a_2, \dots, a_m . Sea *B* el conjunto de entidades fuertes del que *A* depende. La clave primaria de *B* consiste en los atributos b_1, b_2, \dots, b_n . El conjunto de entidades *A* se representa mediante el esquema de relación denominado *A* con una columna por cada miembro del conjunto:

$$\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$$

Para los esquemas derivados de conjuntos de entidades débiles la combinación de la clave primaria del conjunto de entidades fuertes y del discriminador del conjunto de entidades débiles sirve de clave primaria del esquema. Además de crear una clave primaria, también se crea una restricción de clave externa para la relación *A*, que especifica que los atributos b_1, b_2, \dots, b_n hacen referencia a la clave primaria de la relación *B*. La restricción de clave externa garantiza que por cada tupla que represente a una entidad débil exista la tupla correspondiente que representa a la entidad fuerte correspondiente.

Como ilustración, considérese el conjunto de entidades *pago* del diagrama E-R de la Figura 6.19. Este conjunto de entidades tiene tres atributos: *número_pago*, *fecha_pago* e *importe_pago*. La clave primaria del conjunto de entidades *préstamo*, de la que *pago* depende, es *número_préstamo*. Por tanto, *pago* se representa mediante un esquema con cuatro atributos:

$$pago = (\underline{número_préstamo}, \underline{número_pago}, fecha_pago, importe)$$

La clave primaria consiste en la clave primaria de *préstamo* y el discriminador de *pago* (*número_pago*). También se crea una restricción de clave externa para el esquema *pago*, con el atributo *número_pago* que hace referencia a la clave primaria del esquema *préstamo*.

6.9.3 Representación de los conjuntos de relaciones

Sea *R* un conjunto de relaciones, sea a_1, a_2, \dots, a_m el conjunto de atributos formado por la unión de las claves primarias de cada uno de los conjuntos de entidades que participan en *R*, y sean b_1, b_2, \dots, b_n los atributos descriptivos de *R* (si los hay). El conjunto de relaciones se representa mediante el esquema de relación *R*, con un atributo por cada uno de los miembros del conjunto:

$$\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$$

Ya se ha descrito, en el Apartado 6.3.2.2, la manera de escoger la clave primaria de un conjunto de relaciones binarias. Como se vio en ese apartado, tomar todos los atributos de las claves primarias de todos los conjuntos de entidades primarias sirve para identificar una tupla concreta pero, para los conjuntos de relaciones uno a uno, varios a uno y uno a varios, esto resulta un conjunto de atributos mayor del que hace falta en la clave primaria. En vez de eso, la clave primaria se escoge de la manera siguiente:

- Para las relaciones binarias varios a varios la unión de los atributos de clave primaria de los conjuntos de entidades participantes pasa a ser la clave primaria.
- Para los conjuntos de relaciones binarias uno a uno la clave primaria de cualquiera de los conjuntos de entidades puede escogerse como clave primaria de la relación. La elección del conjunto de entidades de entre los relacionados por el conjunto de relaciones puede realizarse de manera arbitraria.
- Para los conjuntos de relaciones binarias varios a uno o uno a varios la clave primaria del conjunto de entidades de la parte “varios” de la relación sirve de clave primaria.
- Para los conjuntos de relaciones n -arias sin flechas en los segmentos la unión de los atributos de clave primaria de los conjuntos de entidades participantes pasa a ser la clave primaria.
- Para los conjuntos de relaciones n -arias con una flecha en uno de los segmentos las claves primarias de los conjuntos de entidades que no están en el lado “flecha” del conjunto de relaciones sirven de clave primaria del esquema. Recuérdese que sólo se permite una flecha saliente por conjunto de relaciones.

También se crean restricciones de clave externa para la relación R de la manera siguiente. Para cada conjunto de entidades E_i relacionado con el conjunto de relaciones R se crea una restricción de clave primaria de la relación R , con los atributos de R que eran atributos de clave primaria de E que hacen referencia a la clave primaria de la relación que representa E_i .

Como ejemplo, considérese el conjunto de relaciones *prestatario* del diagrama E-R de la Figura 6.7. Este conjunto de relaciones implica a los dos conjuntos de entidades siguientes:

- *cliente*, con la clave primaria *id_cliente*
- *préstamo*, con la clave primaria *número_préstamo*

Como el conjunto de relaciones no tiene ningún atributo, el esquema *prestatario* tiene dos atributos:

$$\text{prestatario} = (\underline{\text{id_cliente}}, \underline{\text{número_préstamo}})$$

La clave primaria de la relación *prestatario* es la unión de los atributos de clave primaria de *cliente* y de *préstamo*. También se crean dos restricciones de clave externa para la relación *prestatario*, con el atributo *id_cliente* que hace referencia a la clave primaria de *cliente* y el atributo *número_préstamo* que hace referencia a la clave primaria de *préstamo*.

6.9.3.1 Redundancia de los esquemas

Los conjuntos de relaciones que enlazan los conjuntos de entidades débiles con el conjunto correspondiente de entidades fuertes se tratan de manera especial. Como se hizo notar en el Apartado 6.6, estas relaciones son varios a uno y no tienen atributos descriptivos. Además, la clave primaria de los conjuntos de entidades débiles incluye la clave primaria de los conjuntos de entidades fuertes. En el diagrama E-R de la Figura 6.19, el conjunto de entidades débiles *pago* depende del conjunto de entidades fuertes *préstamo* a través del conjunto de relaciones *pago_préstamo*. La clave primaria de *pago* es *número_préstamo*, *número_pago* y la clave primaria de *préstamo* es *número_préstamo*. Como *pago_préstamo* no tiene atributos descriptivos, el esquema *pago_préstamo* tiene dos atributos, *número_préstamo* y *número_pago*. El esquema del conjunto de entidades *pago* tiene cuatro atributos, *número_préstamo*, *número_pago*, *fecha_pago* e *importe_pago*. Cada combinación (*número_préstamo*, *número_pago*) de una relación de *pago_préstamo* también se halla presente en el esquema de relación *pago*, y viceversa. Por tanto, el esquema *pago_préstamo* es redundante. En general, el esquema de los conjuntos de relaciones que enlazan los conjuntos de entidades débiles con su conjunto correspondiente de entidades fuertes es redundante y no hace falta que esté presente en el diseño de la base de datos relacional basado en el diagrama E-R.

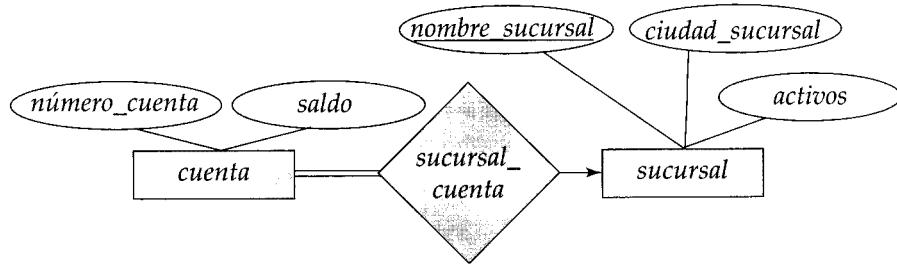


Figura 6.27 Diagrama E-R.

6.9.3.2 Combinación de esquemas

Considérese un conjunto AB de relaciones varios a uno del conjunto de entidades A al conjunto de entidades B . Usando el esquema de construcción de esquemas de relación descrito previamente se consiguen tres esquemas: A , B y AB . Supóngase, además, que la participación de A en la relación es total; es decir, todas las entidades a del conjunto de entidades A deben participar en la relación AB . Entonces se pueden combinar los esquemas A y AB para formar un solo esquema consistente en la unión de los atributos de los dos esquemas.

Como ilustración, considérese el diagrama E-R de la Figura 6.27. La línea doble del diagrama E-R indica que la participación de $cuenta$ en $cuenta_sucursal$ es total. Por tanto, no puede haber ninguna cuenta que no esté asociada a alguna sucursal. Además, el conjunto de relaciones $cuenta_sucursal$ es varios a uno de $cuenta$ a $sucursal$. Por lo tanto, se puede combinar el esquema de $cuenta_sucursal$ con el esquema de $cuenta$ y sólo se necesitan los dos esquemas siguientes:

- $cuenta = (\text{número_cuenta}, \text{saldo}, \text{nombre_sucursal})$
- $sucursal = (\text{nombre_sucursal}, \text{ciudad_sucursal}, \text{activos})$

En el caso de las relaciones uno a uno, el esquema de relación del conjunto de relaciones puede combinarse con el esquema de cualquiera de los conjuntos de entidades.

Se pueden combinar esquemas aunque la participación sea parcial, usando los valores nulos; en el ejemplo anterior se almacenarían valores nulos para el atributo $nombre_sucursal$ de las cuentas que no tengan sucursal asociada.

La clave primaria del esquema combinado es la clave primaria del conjunto de entidades en cuyo esquema se ha fusionado el conjunto de relaciones. En el ejemplo anterior, la clave primaria es $número_cuenta$.

En circunstancias normales el esquema que representa al conjunto de relaciones habría tenido restricciones de clave externa que harían referencia a cada uno de los conjuntos de entidades que participan en el conjunto de relaciones. En este caso se ha descartado la restricción que hace referencia al conjunto de entidades en cuyo esquema se ha fusionado el conjunto de relaciones y se ha añadido las demás restricciones de clave externa al esquema combinado. En el ejemplo anterior, se descarta la restricción de clave externa que hace referencia a $cuenta$, pero se conserva la restricción de clave externa en la que $nombre_sucursal$ hace referencia a $sucursal$ como restricción del esquema combinado $cuenta$.

6.9.4 Atributos compuestos y multivalorados

Los atributos compuestos se tratan mediante la creación de un atributo diferente para cada uno de los atributos componentes; no se crea ningún atributo para el atributo compuesto propiamente dicho. Supóngase que $dirección$ es un atributo compuesto del conjunto de entidades $cliente$ y que los componentes de $dirección$ son $calle$ y $ciudad$. El esquema generado a partir de $cliente$ contiene las columnas $calle_dirección$ y $ciudad_dirección$; no hay ningún atributo ni esquema para $dirección$. Este asunto se vuelve a tratar en el Apartado 7.2.

Se ha visto que en los diagramas E-R los atributos se suelen asignar directamente a las columnas de los esquemas de relación correspondientes. Los atributos multivalorados, sin embargo, son una excepción; para estos atributos se crean esquemas de relación nuevos.

Para cada atributo multivalorado M se crea un esquema de relación E con un atributo A que corresponde a M y a los atributos correspondientes a la clave primaria del conjunto de entidades o de relaciones del que M es atributo.

Como ilustración, considérese el diagrama E-R de la Figura 6.25. El diagrama incluye el conjunto de entidades *empleado* con el atributo multivalorado *nombre_subordinado*. La clave primaria de *empleado* es *id_empleado*. Para este atributo multivalorado se crea el esquema de relación

$$\underline{\text{nombre_subordinado}} \ (\underline{\text{id_empleado}}, \underline{\text{nombre_subordinado}})$$

En este esquema cada subordinado de un empleado se representa mediante una sola tupla de la relación. Por tanto, si se tuviera un empleado con *id_empleado* 12-234 y los subordinados Martín y María, la relación *nombre_subordinado* tendría dos tuplas, (12-234, Martín) y (12-234, María).

Se crea una clave primaria del esquema de la relación consistente en todos los atributos del esquema. En el ejemplo anterior, la clave primaria consiste en todos los atributos de la relación *nombre_subordinado*.

Además, se crea una clave externa para el esquema de la relación, con el atributo generado a partir de la clave primaria del conjunto de entidades que hace referencia a la relación generada a partir del conjunto de entidades. En el ejemplo anterior, la restricción sería que el atributo *id_empleado* hace referencia a la relación *empleado*.

6.9.5 Representación de la generalización

Existen dos métodos diferentes para designar los esquemas de relación de los diagramas E-R que incluyen generalización. Aunque en esta discusión se hace referencia a la generalización de la Figura 6.20, se simplifica incluyendo sólo la primera capa de los conjuntos de entidades de nivel inferior—es decir, *empleado* y *cliente*. Se da por supuesto que *id_persona* es la clave primaria de *persona*.

1. Se crea un esquema para el conjunto de entidades de nivel superior. Para cada conjunto de entidades de nivel inferior se crea un esquema que incluye un atributo para cada uno de los atributos de ese conjunto de entidades más un atributo por cada atributo de la clave primaria del conjunto de entidades de nivel superior. Así, para el diagrama E-R de la Figura 6.20, se tienen tres esquemas:

$$\begin{aligned} \underline{\text{persona}} &= (\underline{\text{id_persona}}, \underline{\text{nombre}}, \underline{\text{calle}}, \underline{\text{ciudad}}) \\ \underline{\text{empleado}} &= (\underline{\text{id_persona}}, \underline{\text{sueldo}}) \\ \underline{\text{cliente}} &= (\underline{\text{id_persona}}, \underline{\text{calificación_crediticia}}) \end{aligned}$$

Los atributos de clave primaria del conjunto de entidades de nivel superior pasan a ser atributos de clave primaria del conjunto de entidades de nivel superior y de todos los conjuntos de entidades de nivel inferior. En el ejemplo anterior se pueden ver subrayados.

Además, se crean restricciones de clave externa para los conjuntos de entidades de nivel inferior, con sus atributos de clave primaria que hacen referencia a la clave primaria de la relación creada a partir del conjunto de entidades de nivel superior. En el ejemplo anterior, el atributo *id_persona* de *empleado* haría referencia a la clave primaria de *persona*, y algo parecido puede decirse de *cliente*.

2. Es posible una representación alternativa si la generalización es disjunta y completa—es decir, si no hay ninguna entidad miembro de dos conjuntos de entidades de nivel inferior directamente por debajo de un conjunto de entidades de nivel superior, y si todas las entidades del conjunto de entidades de nivel superior también pertenece a uno de los conjuntos de entidades de nivel inferior. En este caso no se crea un esquema para el conjunto de entidades de nivel superior. En vez de eso, para cada conjunto de entidades de nivel inferior se crea un esquema que incluye un atributo por cada atributo de ese conjunto de entidades más un atributo por *cada* atributo del conjunto de entidades de nivel superior. Entonces, para el diagrama E-R de la Figura 6.20, se tienen dos esquemas:

$$\begin{aligned} empleado &= (\underline{id_persona}, \underline{nombre}, calle, ciudad, sueldo) \\ cliente &= (\underline{id_persona}, \underline{nombre}, calle, ciudad, \underline{calificación_crediticia}) \end{aligned}$$

Estos dos esquemas tienen *id_persona*, que es el atributo de clave primaria del conjunto de entidades de nivel superior *persona*, como clave primaria.

Un inconveniente del segundo método es la definición de las restricciones de clave externa. Para ilustrar el problema, supóngase que se tiene un conjunto de relaciones *R* que implica al conjunto de entidades *persona*. Con el primer método, al crear un esquema de relación *R* a partir del conjunto de relaciones, también se define una restricción de clave externa para *R*, que hace referencia al esquema *persona*. Desafortunadamente, con el segundo método no se tiene una única relación a la que pueda hacer referencia la restricción de clave externa de *R*. Para evitar este problema, hay que crear un esquema de relación *persona* que contenga, al menos, los atributos de clave primaria de la entidad *persona*.

Si se usara el segundo método para una generalización solapada, algunos valores se almacenarían varias veces, de manera innecesaria. Por ejemplo, si una persona es a la vez empleado y cliente, los valores de *calle* y de *ciudad* se almacenarían dos veces. Si la generalización no fuera completa—es decir, si alguna persona no fuera ni empleado ni cliente—entonces haría falta un esquema para representar a esas personas.

6.9.6 Representación de la agregación

El diseño de esquemas para los diagramas E-R que incluyen agregación es sencillo. Considérese el diagrama de la Figura 6.22. El esquema del conjunto de relaciones *dirige* entre la agregación de *trabaja_en* y el conjunto de entidades *director* incluye un atributo para cada atributo de las claves primarias del conjunto de entidades *director* y del conjunto de relaciones *trabaja_en*. También incluye un atributo para los atributos descriptivos, si los hay, del conjunto de relaciones *dirige*. Por tanto, se transforman los conjuntos de relaciones y de entidades de la entidad agregada siguiendo las reglas que se han definido anteriormente.

Las reglas que se han visto anteriormente para la creación de restricciones de clave primaria y de clave externa para los conjuntos de relaciones se pueden aplicar también a los conjuntos de relaciones que incluyen agregación, tratando la agregación como cualquier otra entidad. La clave primaria de la agregación es la clave primaria del conjunto de relaciones que la define. No hace falta ninguna relación más para que represente la agregación; en vez de eso, se usa la relación creada a partir de la relación definidora.

6.9.7 Esquemas relacionales para la entidad bancaria

En la Figura 6.25 se mostró el diagrama E-R de una entidad bancaria. El conjunto correspondiente de esquemas de relación, generado mediante las técnicas ya descritas en este apartado, se muestra a continuación. La clave primaria de cada esquema de relación se denota mediante el subrayado.

- Esquemas derivados de entidades fuertes:

$$\begin{aligned} sucursal &= (\underline{nombre_sucursal}, \underline{ciudad_sucursal}, activos) \\ cliente &= (\underline{id_cliente}, \underline{nombre_cliente}, calle_cliente, ciudad_cliente) \\ préstamo &= (\underline{número_préstamo}, importe) \\ cuenta &= (\underline{número_cuenta}, saldo) \\ empleado &= (\underline{id_empleado}, \underline{nombre_empleado}, número_teléfono, fecha_contratación) \end{aligned}$$

- Esquemas derivados de atributos multivalorados (no se representan los atributos derivados). Se definen en una vista o en una función definida especialmente:

$$\underline{nombre_subordinado} = (\underline{id_empleado}, \underline{nombre_subordinado})$$

- Esquemas derivados de conjuntos de relaciones que implican a conjuntos de entidades fuertes:

$\text{sucursal_cuenta} = (\underline{\text{número_cuenta}}, \underline{\text{nombre_sucursal}})$
 $\text{sucursal_préstamo} = (\underline{\text{número_préstamo}}, \underline{\text{nombre_sucursal}})$
 $\text{prestatario} = (\underline{\text{id_cliente}}, \underline{\text{número_préstamo}})$
 $\text{impositor} = (\underline{\text{id_cliente}}, \underline{\text{número_cuenta}})$
 $\text{asesor} = (\underline{\text{id_cliente}}, \underline{\text{id_empleado}}, \underline{\text{tipo}})$
 $\text{trabaja_para} = (\underline{\text{id_empleado_trabajador}}, \underline{\text{id_empleado_jefe}})$

- Esquemas derivados de conjuntos de entidades débiles (recuérdese que se probó en el Apartado 6.9.3.1 que el esquema pago_préstamo es redundante):

$\text{pago} = (\underline{\text{número_préstamo}}, \underline{\text{número_pago}}, \underline{\text{fecha_pago}}, \underline{\text{importe}})$

- Esquemas derivados de relaciones ES (se ha escogido la primera de las dos opciones presentadas en el Apartado 6.9.5 para permitir las cuentas que no son ni de ahorro ni corrientes):

$\text{cuenta_ahorro} = (\underline{\text{número_cuenta}}, \underline{\text{tasa_interés}})$
 $\text{cuenta_corriente} = (\underline{\text{número_cuenta}}, \underline{\text{importe_descubierto}})$

Se deja como ejercicio la creación de las restricciones de clave externa adecuadas para las relaciones anteriores.

6.10 Otros aspectos del diseño de bases de datos

La explicación sobre el diseño de esquemas dada en este capítulo puede crear la falsa impresión de que el diseño de esquemas es el único componente del diseño de bases de datos. En realidad, hay otras consideraciones que se tratarán con más profundidad en capítulos posteriores y que se describirán brevemente a continuación.

6.10.1 Restricciones de datos y diseño de bases de datos relacionales

Se ha visto gran variedad de restricciones de datos que pueden expresarse mediante SQL, como las restricciones de clave primaria, las de clave externa, las restricciones **check**, los asertos y los disparadores. Las restricciones tienen varios propósitos. El más evidente es la automatización de la conservación de la consistencia. Al expresar las restricciones en el lenguaje de definición de datos de SQL, el diseñador puede garantizar que el propio sistema de bases de datos haga que se cumplan las restricciones. Esto es más digno de confianza que dejar que cada programa haga cumplir las restricciones por su cuenta. También ofrece una ubicación central para la actualización de las restricciones y la adición de otras nuevas.

Otra ventaja de definir explícitamente las restricciones es que algunas restricciones resultan especialmente útiles en el diseño de esquemas de bases de datos relacionales. Si se sabe, por ejemplo, que el número de DNI identifica de manera única a cada persona, se puede usar el número de DNI de una persona para vincular los datos relacionados con esa persona aunque aparezcan en varias relaciones. Compárese esta posibilidad, por ejemplo, con el color de ojos, que no es un identificador único. El color de ojos no se puede usar para vincular los datos correspondientes a una persona concreta en varias relaciones, ya que los datos de esa persona no se podrían distinguir de los datos de otras personas con el mismo color de ojos.

En el Apartado 6.9 se generó un conjunto de esquemas de relación para un diseño E-R dado mediante las restricciones especificadas en el diseño. En el Capítulo 7 se formaliza esta idea y otras relacionadas y se muestra la manera en que puede ayudar al diseño de esquemas de bases de datos relacionales. El enfoque formal del diseño de bases de datos relacionales permite definir de manera precisa la bondad de cada diseño y mejorar los diseños malos. Se verá que el proceso de comenzar con un diseño entidad-relación y generar mediante algoritmos los esquemas de relación a partir de ese diseño es una buena manera de comenzar el proceso de diseño.

Las restricciones de datos también resultan útiles para determinar la estructura física de los datos. Puede resultar útil almacenar físicamente próximos en el disco los datos que están estrechamente relacionados entre sí, de modo que se mejore la eficiencia del acceso a disco. Algunas estructuras de índices funcionan mejor cuando el índice se crea sobre una clave primaria.

La aplicación de las restricciones se lleva a cabo a un precio potencialmente alto en rendimiento cada vez que se actualiza la base de datos. En cada actualización el sistema debe comprobar todas las restricciones y rechazar las actualizaciones que no las cumplen o ejecutar los disparadores correspondientes. La importancia de la penalización en rendimiento no sólo depende de la frecuencia de actualización, sino también del modo en que se haya diseñado la base de datos. En realidad, la eficiencia de la comprobación de determinados tipos de restricciones es un aspecto importante de la discusión del diseño de esquemas para bases de datos relationales en el Capítulo 7.

6.10.2 Requisitos de uso: consultas y rendimiento

El rendimiento de los sistemas de bases de datos es un aspecto crítico de la mayor parte de los sistemas informáticos empresariales. El rendimiento no sólo tiene que ver con el uso eficiente del hardware de cálculo y de almacenamiento que se usa, sino también con la eficiencia de las personas que interactúan con el sistema y de los procesos que dependen de los datos de las bases de datos.

Existen dos métricas principales para el rendimiento.

- **Productividad**—el número de consultas o actualizaciones (a menudo denominadas *transacciones*) que pueden procesarse en promedio por unidad de tiempo.
- **Tiempo de respuesta**—el tiempo que tarda *una sola* transacción desde el comienzo hasta el final en promedio o en el peor de los casos.

Los sistemas que procesan gran número de transacciones agrupadas por lotes se centran en tener una productividad elevada. Los sistemas que interactúan con personas y los sistemas de tiempo crítico suelen centrarse en el tiempo de respuesta. Estas dos métricas no son equivalentes. La productividad elevada se consigue mediante un elevado uso de los componentes del sistema. Ello puede dar lugar a que algunas transacciones se pospongan hasta el momento en que puedan ejecutarse con mayor eficiencia. Las transacciones pospuestas sufren un bajo tiempo de respuesta.

Históricamente, la mayor parte de los sistemas de bases de datos comerciales se han centrado en la productividad; no obstante, gran variedad de aplicaciones, incluidas las aplicaciones basadas en la Web y los sistemas informáticos para telecomunicaciones necesitan un buen tiempo de respuesta promedio y una cota razonable para el peor tiempo de respuesta que pueden ofrecer.

La comprensión de los tipos de consultas que se espera que sean más frecuentes ayuda al proceso de diseño. Las consultas que implican reuniones necesitan evaluar más recursos que las que no las implican. A veces, cuando se necesita una reunión, puede que el administrador de la base de datos decida crear un índice que facilite la evaluación de la reunión. Para las consultas—tanto si está implicada una reunión como si no—se pueden crear índices para acelerar la evaluación de los predicados de selección (la cláusula **where** de SQL) que sea posible que aparezcan. Otro aspecto de las consultas que afecta a la elección de índices es la proporción relativa de operaciones de actualización y de lectura. Aunque los índices pueden acelerar las consultas, también ralentiza las actualizaciones, que se ven obligadas a realizar un trabajo adicional para mantener la exactitud de los índices.

6.10.3 Requisitos de autorización

Las restricciones de autorización también afectan al diseño de las bases de datos, ya que SQL permite que se autorice el acceso a los usuarios en función de los componentes del diseño lógico de la base de datos. Puede que haga falta descomponer un esquema de relación en dos o más esquemas para facilitar la concesión de derechos de acceso en SQL. Por ejemplo, un registro de empleados puede contener datos relativos a nóminas, funciones de los puestos y prestaciones sanitarias. Como diferentes unidades administrativas de la empresa pueden manejar cada uno de los diferentes tipos de datos, algunos usuarios necesitarán acceso a los datos de las nóminas, mientras se les deniega el acceso a los datos de las funciones de los puestos de trabajo, a los de las prestaciones sanitarias, etc. Si todos esos datos se hallan

en una tabla, la deseada división del acceso, aunque todavía posible mediante el uso de vistas, resulta más complicada. La división de los datos, de este modo, pasa a ser todavía más crítica cuando los datos se distribuyen en varios sistemas de una red informática, un aspecto que se considera en el Capítulo 22.

6.10.4 Flujos de datos y de trabajo

Las aplicaciones de bases de datos suelen formar parte de una aplicación empresarial de mayor tamaño que no sólo interactúa con el sistema de bases de datos, sino también con diferentes aplicaciones especializadas. Por ejemplo, en una compañía manufacturera, puede que un sistema de diseño asistido por computadora (computer-aided design, CAD) ayude al diseño de nuevos productos. Puede que el sistema CAD extraiga datos de la base de datos mediante instrucciones de SQL, procese internamente los datos, quizás interactuando con un diseñador de productos, y luego actualice la base de datos. Durante este proceso el control de los datos puede pasar a manos de varios diseñadores de productos y de otras personas. Como ejemplo adicional, considérese un informe de gastos de viaje. Lo crea un empleado que vuelve de un viaje de negocios (posiblemente mediante un paquete de software especial) y luego se envía al jefe de ese empleado, quizás a otros jefes de niveles superiores y, finalmente, al departamento de contabilidad para su pago (momento en el que interactúa con los sistemas informáticos de contabilidad de la empresa).

El término *flujo de trabajo* hace referencia a la combinación de datos y de tareas implicados en procesos como los de los ejemplos anteriores. Los flujos de trabajo interactúan con el sistema de bases de datos cuando se mueven entre los usuarios y los usuarios llevan a cabo sus tareas en el flujo de trabajo. Además de los datos sobre los que opera el flujo de trabajo, puede que la base de datos almacene datos sobre el propio flujo de trabajo, incluidas las tarea que lo conforman y la manera en que se han de hacer llegar a los usuarios. Por tanto, los flujos de trabajo especifican una serie de consultas y de actualizaciones de la base de datos que pueden tenerse en cuenta como parte del proceso de diseño de la base de datos. En otras palabras, el modelado de la empresa no sólo exige la comprensión de la semántica de los datos, sino también la de los procesos comerciales que los usan.

6.10.5 Otros problemas del diseño de bases de datos

El diseño de bases de datos no suele ser una actividad que se pueda dar por acabada. Las necesidades de las organizaciones evolucionan continuamente, y los datos que necesitan almacenar también evolucionan en consonancia. Durante las fases iniciales del diseño de la base de datos, o durante el desarrollo de las aplicaciones, puede que el diseñador de la base de datos se dé cuenta de que hacen falta cambios en el nivel del esquema conceptual, lógico o físico. Los cambios del esquema pueden afectar a todos los aspectos de la aplicación de bases de datos. Un buen diseño de bases de datos se anticipa a las necesidades futuras de la organización y el diseño se lleva a cabo de manera que se necesiten modificaciones mínimas a medida que evolucionen las necesidades de la organización.

Es importante distinguir entre las restricciones fundamentales y las que se anticipa que puedan cambiar. Por ejemplo, la restricción de que cada identificador de cliente identifique a un solo cliente es fundamental. Por otro lado, el banco puede tener la norma de que cada cliente sólo pueda tener una cuenta, lo que puede cambiar más adelante. Un diseño de la base de datos que sólo permita una cuenta por cliente necesitaría cambios importantes si el banco cambiase su normativa. Esos cambios no deben exigir modificaciones importantes en el diseño de la base de datos.

Además, es probable que la empresa a la que sirve la base de datos interactúe con otras empresas y, por tanto, puede que tengan que interactuar varias bases de datos. La conversión de los datos entre esquemas diferentes es un problema importante en las aplicaciones del mundo real. Se han propuesto diferentes soluciones para este problema. El modelo de datos XML, que se estudia en el Capítulo 10, se usa mucho para representar los datos cuando se intercambian entre diferentes aplicaciones.

Finalmente, merece la pena destacar que el diseño de bases de datos es una actividad orientada a los seres humanos en dos sentidos: los usuarios finales son personas (aunque se sitúe alguna aplicación entre la base de datos y los usuarios finales) y el diseñador de la base de datos debe interactuar intensamente con los expertos en el dominio de la aplicación para comprender los requisitos de datos de la aplicación. Todas las personas involucradas con los datos tiene necesidades y preferencias que se deben tener en cuenta para que el diseño y la implantación de la base de datos tengan éxito en la empresa.

6.11 El lenguaje de modelado unificado UML**

Los diagramas entidad-relación ayudan a modelar el componente de representación de datos de los sistemas de software. La representación de datos, sin embargo, sólo forma parte del diseño global del sistema. Otros componentes son los modelos de interacción del usuario con el sistema, la especificación de los módulos funcionales del sistema y su interacción, etc. El **lenguaje de modelado unificado** (Unified Modeling Language, UML) es una norma desarrollada bajo los auspicios del Grupo de Administración de Objetos (Object Management Group, OMG) para la creación de especificaciones de diferentes componentes de los sistemas de software. Algunas de las partes de UML son:

- **Diagramas de clase.** Los diagramas de clase son parecidos a los diagramas E-R. Más adelante en este apartado se mostrarán algunas características de los diagramas de clase y del modo en que se relacionan con los diagramas E-R.
- **Diagramas de caso de uso.** Los diagramas de caso de uso muestran la interacción entre los usuarios y el sistema, en especial los pasos de las tareas que llevan a cabo los usuarios (como retirar dinero o matricularse en una asignatura).
- **Diagramas de actividad.** Los diagramas de actividad describen el flujo de tareas entre los diferentes componentes del sistema.
- **Diagramas de implementación.** Los diagramas de implementación muestran los componentes del sistema y sus interconexiones, tanto en el nivel de los componentes de software como en el de hardware.

Aquí no se pretende ofrecer un tratamiento detallado de las diferentes partes del UML. Véanse las notas bibliográficas para encontrar referencias sobre UML. En vez de eso, se ilustrarán algunas características de la parte de UML que se relaciona con el modelado de datos mediante ejemplos.

La Figura 6.28 muestra varios constructores de diagramas E-R y sus constructores equivalentes de diagramas de clases de UML. Más adelante se describen estos constructores. UML muestra los conjuntos de entidades como cuadros y, a diferencia de E-R, muestra los atributos dentro de los cuadros en lugar de como elipses separadas. UML modela realmente objetos, mientras que E-R modela entidades. Los objetos son como entidades y tienen atributos, pero también proporcionan un conjunto de funciones (denominadas métodos) que se pueden invocar para calcular valores con base en los atributos de los objetos, o para actualizar el propio objeto. Los diagramas de clases pueden describir métodos además de atributos. Los objetos se tratan en el Capítulo 9.

Los conjuntos de relaciones binarias se representan en UML dibujando simplemente una línea que conecte los conjuntos de entidades. El nombre del conjunto de relaciones se escribe junto a la línea. También se puede especificar el rol que desempeña cada conjunto de entidades en un conjunto de relaciones escribiendo el nombre del rol sobre la línea, junto al conjunto de entidades. De manera alternativa, se puede escribir el nombre del conjunto de relaciones en un recuadro, junto con los atributos del conjunto de relaciones, y conectar el recuadro con una línea discontinua a la línea que describe el conjunto de relaciones. Este recuadro se puede tratar entonces como un conjunto de entidades, de la misma forma que la agregación en los diagramas E-R, y puede participar en relaciones con otros conjuntos de entidades.

Desde la versión 1.3 de UML, UML soporta las relaciones no binarias, usando la misma notación de rombos usada en los diagramas E-R. Las relaciones no binarias no se podían representar directamente en versiones anteriores de UML—había que convertirlas en relaciones binarias usando la técnica que se vio en el Apartado 6.5.3.

Las restricciones de cardinalidad se especifican en UML de la misma forma que en los diagramas E-R, de la forma $i..s$, donde i denota el número mínimo y s el máximo de relaciones en que puede participar cada entidad. Sin embargo, hay que ser consciente de que la ubicación de las restricciones es exactamente la contraria que en los diagramas E-R, como se muestra en la Figura 6.28. La restricción $0..*$ en el lado $E2$ y $0..1$ en el lado $E1$ significa que cada entidad $E2$ puede participar, a lo sumo, en una relación, mientras que cada entidad $E1$ puede participar en varias relaciones; en otras palabras, la relación es varios a uno de $E2$ a $E1$.

Los valores aislados como 1 o * se pueden escribir en los arcos; el valor 1 sobre un arco se trata como equivalente de $1..1$, mientras que * es equivalente a $0..*$.

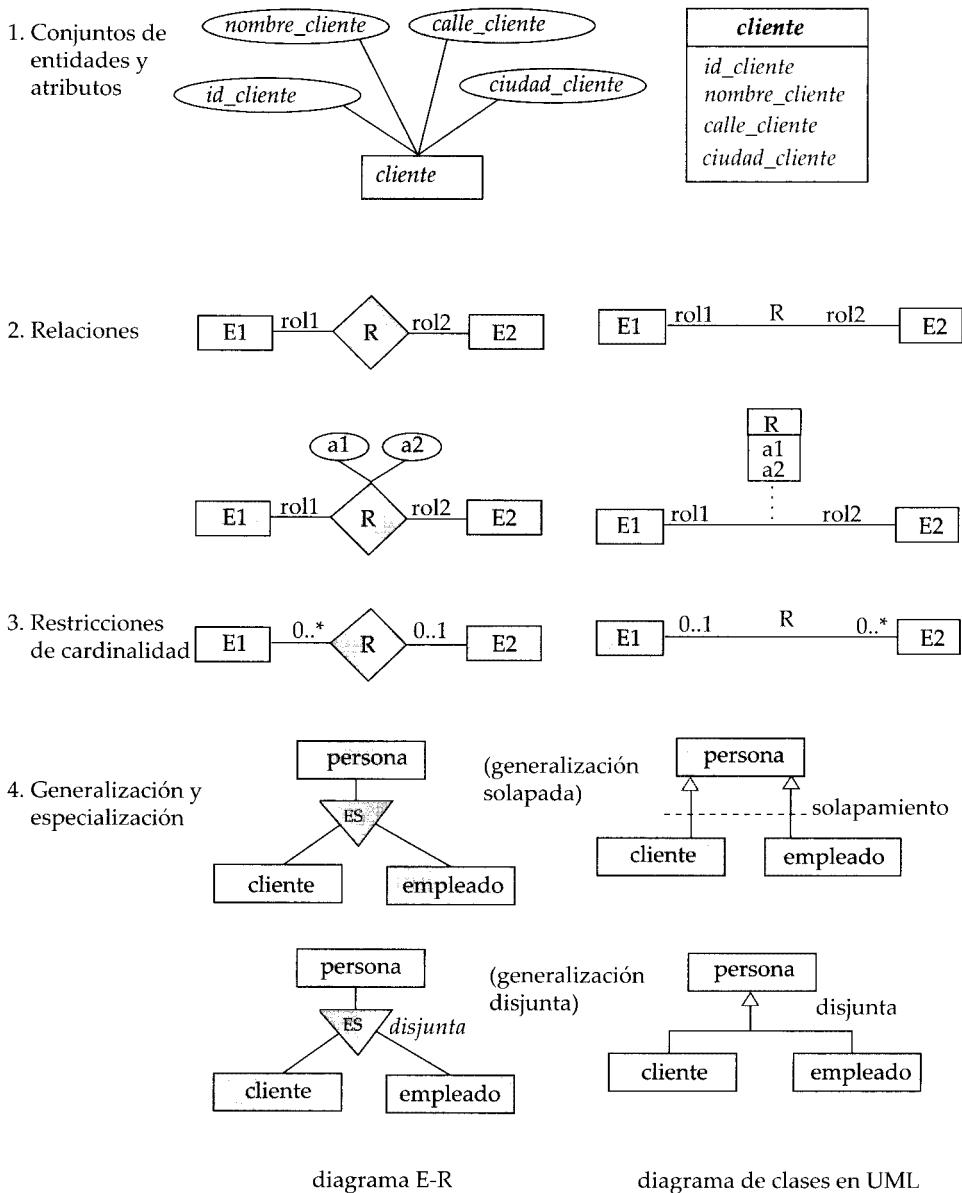


Figura 6.28 Símbolos usados en la notación de diagramas de clases de UML.

La generalización y la especialización se representan en UML conectando conjuntos de entidades mediante una línea con un triángulo al final correspondiente al conjunto de entidades más general. Por ejemplo, el conjunto de entidades *persona* es una generalización de *cliente* y de *empleado*. Los diagramas UML también pueden representar explícitamente las restricciones de la condición de disjunción y de solapamiento de las generalizaciones. La Figura 6.28 muestra generalizaciones disjuntas y solapadas de *cliente* y *empleado* a *persona*. Recuérdese que si la generalización de *cliente/empleado* a *persona* es disjunta, eso significa que nadie puede ser a la vez *cliente* y *empleado*. Una generalización solapada permite que una persona sea tanto *cliente* como *empleado*.

Los diagramas de clases de UML incluyen otras notaciones que no se corresponden con las notaciones E-R que se han visto. Por ejemplo, una línea entre dos conjuntos de entidades con un rombo en un extremo especifica que la entidad en el extremo del rombo contiene a la otra (la inclusión se denomina “agregación” en la terminología de UML). Por ejemplo, la entidad *vehículo* puede contener una entidad *motor*. Los diagramas de clases de UML también ofrecen notaciones para representar características del lenguaje orientadas a objetos, como las anotaciones públicas o privadas de los miembros de la clase e

interfaces (esto debe resultarle familiar a cualquiera que conozca los lenguajes Java o C#). Véanse las referencias en las notas bibliográficas para obtener más información sobre los diagramas de clases de UML.

6.12 Resumen

- El diseño de bases de datos supone principalmente el diseño del esquema de la base de datos. El modelo de datos **entidad-relación (E-R)** es un modelo de datos muy usado para el diseño de bases de datos. Ofrece una representación gráfica adecuada para ver los datos, las relaciones y las restricciones.
- El modelo está pensado principalmente para el proceso de diseño de la base de datos. Se desarrolló para facilitar el diseño de bases de datos al permitir la especificación de un **esquema de la empresa**. Este esquema representa la estructura lógica general de la base de datos. Esta estructura general se puede expresar gráficamente mediante un **diagrama E-R**.
- Una **entidad** es un objeto que existe en el mundo real y es distingible de otros objetos. Esta distinción se expresa asociando a cada entidad un conjunto de atributos que describen el objeto.
- Una **relación** es una asociación entre diferentes entidades. Un **conjunto de relaciones** es una colección de entidades del mismo tipo, y un **conjunto de entidades** es una colección de entidades del mismo tipo.
- Una **superclave** de un conjunto de entidades es un conjunto de uno o más atributos que, tomados en conjunto, permiten identificar únicamente una entidad del conjunto de entidades. Se elige una superclave mínima para cada conjunto de entidades de entre sus superclaves; la superclave mínima se denomina **clave primaria** del conjunto de entidades. Análogamente, un conjunto de relaciones es un conjunto de uno o más atributos que, tomados en conjunto, permiten identificar únicamente una relación del conjunto de relaciones. De igual forma se elige una superclave mínima para cada conjunto de relaciones de entre todas sus superclaves; ésa es la clave primaria del conjunto de relaciones.
- La **correspondencia de cardinalidades** expresa el número de entidades con las que otra entidad se puede asociar mediante un conjunto de relaciones.
- Un conjunto de entidades que no tiene suficientes atributos para formar una clave primaria se denomina **conjunto de entidades débiles**. Un conjunto de entidades que tiene una clave primaria se denomina **conjunto de entidades fuertes**.
- La **especialización** y la **generalización** definen una relación de inclusión entre un conjunto de entidades de nivel superior y uno o más conjuntos de entidades de nivel inferior. La especialización es el resultado de tomar un subconjunto de un conjunto de entidades de nivel superior para formar un conjunto de entidades de inferior. La generalización es el resultado de tomar la unión de dos o más conjuntos disjuntos de entidades (de nivel inferior) para producir un conjunto de entidades de nivel superior. Los atributos de los conjuntos de entidades de nivel superior heredan los conjuntos de entidades de nivel inferior.
- La **agregación** es una abstracción en la que los conjuntos de relaciones (junto con sus conjuntos de entidades asociados) se tratan como conjuntos de entidades de nivel superior y puede participar en las relaciones.
- Las diferentes características del modelo E-R ofrecen al diseñador de bases de datos numerosas opciones a la hora de representar lo mejor posible la empresa que se modela. Los conceptos y los objetos pueden, en ciertos casos, representarse mediante entidades, relaciones o atributos. Ciertos aspectos de la estructura global de la empresa se pueden describir mejor usando los conjuntos de entidades débiles, la generalización, la especialización o la agregación. A menudo, el diseñador debe sopesar las ventajas de un modelo simple y compacto frente a las de otro más preciso pero más complejo.

- El diseño de una base de datos especificado en un diagrama E-R se puede representar mediante un conjunto de esquemas de relación. Para cada conjunto de entidades y para cada conjunto de relaciones de la base de datos hay un solo esquema de relación al que se le asigna el nombre del conjunto de entidades o de relaciones correspondiente. Esto forma la base para la obtención del diseño de la base de datos relacional a partir del E-R.
- El **lenguaje de modelado unificado (UML)** ofrece un medio gráfico de modelar los diferentes componentes de los sistemas de software. El componente diagrama de clases de UML se basa en los diagramas E-R. Sin embargo, hay algunas diferencias entre los dos que se deben tener presentes.

Términos de repaso

- Modelo de datos entidad-relación.
- Entidad.
- Conjunto de entidades.
- Relación y conjunto de relaciones.
- Rol.
- Conjunto de relaciones recursivo.
- Atributos descriptivos.
- Conjunto de relaciones binarias.
- Grado de un conjunto de relaciones.
- Atributos.
- Dominio.
- Atributos simples y compuestos.
- Atributos monovalorados y multivalorados.
- Valor nulo.
- Atributo derivado.
- Superclave, clave candidata y clave primaria.
- Correspondencia de cardinalidad:
 - Relación uno a uno.
 - Relación uno a varios.
 - Relación varios a uno.
 - Relación varios a varios.
- Participación:
 - Total.
 - Parcial.
- Conjuntos de entidades débiles y fuertes:
 - Atributos discriminantes.
 - Relaciones identificadoras.
- Especialización y generalización:
 - Superclase y subclase.
 - Herencia de atributos.
 - Herencia simple y múltiple.
 - Pertenencia definida por condición y definida por el usuario.
 - Generalización disjunta y solapada.
- Restricción de completitud:
 - Generalización total y parcial.
- Agregación.
- Diagrama E-R.
- Lenguaje de modelado unificado (UML).

Ejercicios prácticos

- 6.1 Constrúyase un diagrama E-R para una compañía de seguros de coches cuyos clientes poseen uno o más coches cada uno. Cada coche tiene asociado un valor que va de cero al número de accidentes registrados.
- 6.2 La secretaría de una universidad conserva datos acerca de las siguientes entidades: (a) asignaturas, incluyendo el número, título, créditos, programa, y requisitos; (b) oferta de asignaturas, incluyendo el número de asignatura, año, semestre, número de sección, profesor(es), horario y aulas; (c) estudiantes, incluyendo identificador de estudiante, nombre y programa; y (d) profesores, incluyendo número de identificación, nombre, departamento y título. Además, la matrícula de los estudiantes en asignaturas y las notas concedidas a los estudiantes en cada asignatura en la que están matriculados se deben modelar adecuadamente.

Constrúyase un diagrama E-R para la secretaría. Documéntense todas las suposiciones que se hagan acerca de las restricciones de asignaciones.

- 6.3** Considérese una base de datos usada para registrar las notas que obtienen los estudiantes en diferentes exámenes de distintas asignaturas.
- Constrúyase un diagrama E-R que modele los exámenes como entidades y utilice una relación ternaria para la base de datos.
 - Constrúyase un diagrama E-R alternativo que sólo utilice una relación binaria entre *estudiantes* y *asignaturas*. Hay que asegurarse de que sólo existe una relación entre cada pareja formada por un estudiante y una asignatura, pero se pueden representar las notas que obtiene cada estudiante en diferentes exámenes de una asignatura.
- 6.4** Diséñese un diagrama E-R para almacenar los logros de su equipo deportivo favorito. Se deben almacenar los partidos jugados, el resultado de cada partido, los jugadores de cada partido y las estadísticas de cada jugador en cada partido. Las estadísticas resumidas se deben representar como atributos derivados.
- 6.5** Considérese un diagrama E-R en el que el mismo conjunto de entidades aparezca varias veces. ¿Por qué permitir esa redundancia es una mala práctica que se debe evitar siempre que sea posible?
- 6.6** Considérese la base de datos de una universidad para la planificación de las aulas para los exámenes finales. Esta base de datos se puede modelar como el conjunto de entidades único *examen*, con los atributos *nombre_asignatura*, *número_sección*, *número_aula* y *hora*. Alternativamente se pueden definir uno o más conjuntos de entidades adicionales, junto con conjuntos de relaciones para sustituir algunos de los atributos del conjunto de entidades *examen*, como
- *asignatura* con atributos *nombre*, *departamento* y *número_a*
 - *sección* con atributos *número_s* y *matriculados*, que es un conjunto de entidades débiles dependiente de *curso*.
 - *aula* con atributos *número_a*, *capacidad* y *edificio*.
- Muéstrese un diagrama E-R que ilustre el uso de los tres conjuntos de entidades adicionales citados.
 - Explíquense las características de la aplicación que influyen en la decisión de incluir o no incluir cada uno de los conjuntos de entidades adicionales.
- 6.7** Cuando se diseña el diagrama E-R para una empresa concreta se tienen varias alternativas para elegir.
- ¿Qué criterio se debe considerar para tomar la decisión adecuada?
 - Diséñense tres diagramas E-R alternativos para representar la secretaría de la universidad del Ejercicio práctico 6.2. Mencíñense las ventajas de cada uno. Arguméntese a favor de una de las alternativas.
- 6.8** Los diagramas E-R se pueden ver como grafos. ¿Qué significa lo siguientes en términos de la estructura del esquema de una empresa?
- El grafo es inconexo.
 - El grafo es acíclico.
- 6.9** Considérese la representación de una relación ternaria mediante relaciones binarias como se describió en el Apartado 6.5.3 e ilustrado en la Figura 6.29 (no se muestran los atributos).
- Muéstrese un ejemplar simple de *E*, *A*, *B*, *C*, *R_A*, *R_B* y *R_C* que no pueda corresponder a ningún ejemplar de *A*, *B*, *C* y *R*.
 - Modifíquese el diagrama E-R de la Figura 6.29b para introducir restricciones que garanticen que cualquier ejemplar *E*, *A*, *B*, *C*, *R_A*, *R_B* y *R_C* que satisfaga las restricciones corresponda a algún ejemplar de *A*, *B*, *C* y *R*.
 - Modifíquese la traducción anterior para manejar las restricciones de participación total sobre las relaciones ternarias.
 - La representación anterior exige que se cree un atributo de clave primaria para *E*. Muéstrese la manera de tratar *E* como un conjunto de entidades débiles de forma que no haga falta ningún atributo de clave primaria.

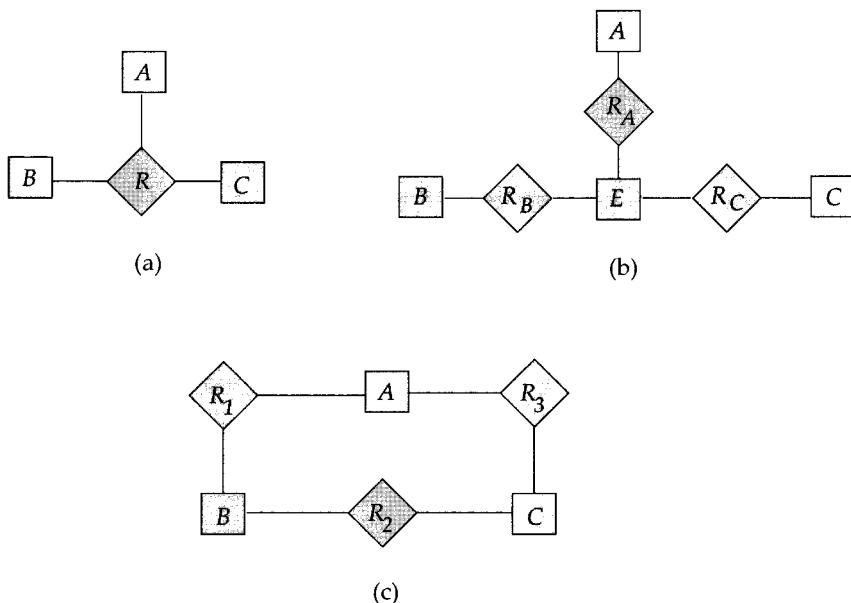


Figura 6.29 Diagrama E-R para el Ejercicio práctico 6.9 y el Ejercicio 6.22.

6.10 Los conjuntos de entidades débiles se pueden convertir en conjuntos de entidades fuertes simplemente añadiendo a sus atributos los atributos de clave primaria del conjunto de entidades identificadoras. Describáse el tipo de redundancia que se produce al hacerlo.

6.11 La Figura 6.30 muestra una estructura reticular de generalización y especialización (no se muestran los atributos). Para los conjuntos de entidades A, B y C explíquese cómo se heredan los atributos desde los conjuntos de entidades de nivel superior X e Y. Explíquese la manera de manejar el caso de que un atributo de X tenga el mismo nombre que algún atributo de Y.

6.12 Considérense dos bancos diferentes que deciden fusionarse. Supóngase que ambos bancos usan exactamente el mismo esquema de bases de datos E-R—el de la Figura 6.25. (Evidentemente, esta suposición es muy poco realista; se considera un caso más realista en el Apartado 22.8). Si el banco fusionado va a tener una sola base de datos, hay varios problemas posibles:

- La posibilidad de que los dos bancos originales tengan sucursales con el mismo nombre.
- La posibilidad de que algunos clientes lo sean de los dos bancos originales.
- La posibilidad de que algún número de préstamo o de cuenta se usara en los dos bancos originales (para diferentes préstamos o cuentas, por supuesto).

Para cada uno de estos posibles problemas describáse el motivo de que existan de hecho la posibilidad de dificultades. Propóngase una solución al problema. Para la solución propuesta, explíquese cualquier cambio que haya que hacer y describáse el efecto que tendrá en el esquema y en los datos.

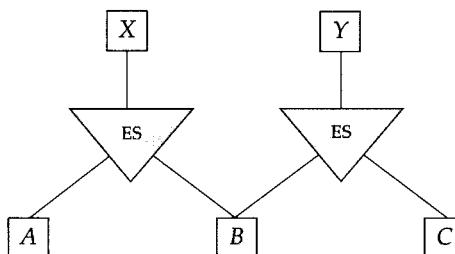


Figura 6.30 Diagrama E-R para el Ejercicio práctico 6.11.

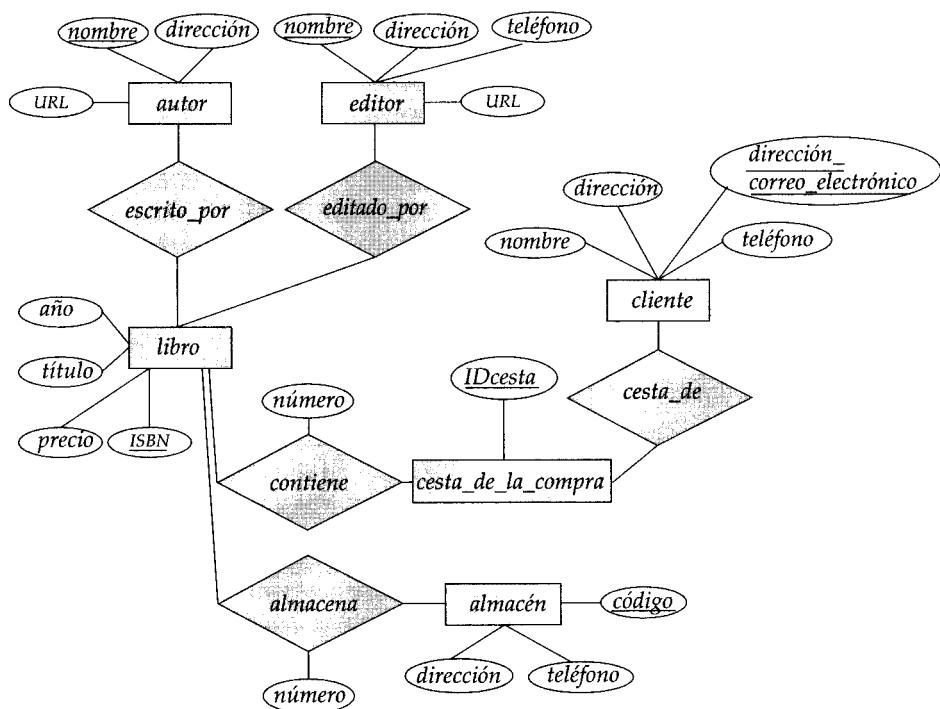


Figura 6.31 Diagrama E-R para el Ejercicio 6.21.

- 6.13 Reconsidérese la situación descrita en el Ejercicio práctico 6.12 bajo la suposición de que un banco está en España y el otro en Portugal. Como antes, los bancos usan el esquema de la Figura 6.25, excepto que el banco portugués usa el número de DNI portugués, mientras que el banco español usa el DNI español para la identificación de los clientes. ¿Qué problemas (además de los identificados en el Ejercicio práctico 6.12) pueden darse en este caso multinacional? ¿Cómo se pueden resolver? Asegúrese de tomar en consideración tanto el esquema como los datos reales al elaborar la respuesta.

Ejercicios

- 6.14 Explíquense las diferencias entre los términos clave primaria, clave candidata y superclave.
- 6.15 Constrúyase un diagrama E-R para un hospital con un conjunto de pacientes y un conjunto de médicos. Asóciense con cada paciente un registro de las diferentes pruebas y exámenes realizados.
- 6.16 Constrúyanse tablas adecuadas para cada uno de los diagramas E-R de los Ejercicios prácticos 6.1 y 6.2.
- 6.17 Extiéndase el diagrama E-R del Ejercicio práctico 6.4 para que almacene la misma información para todos los equipos de una liga.
- 6.18 Explíquense las diferencias entre los conjuntos de entidades débiles y los conjuntos de entidades fuertes.
- 6.19 Se puede convertir cualquier conjunto de entidades débiles en un conjunto de entidades fuertes simplemente añadiendo los atributos apropiados. ¿Por qué, entonces, se tienen conjuntos de entidades débiles?
- 6.20 Defínase el concepto de agregación. Propónganse dos ejemplos en los que este concepto sea útil.
- 6.21 Considérese el diagrama E-R de la Figura 6.31, que modela una librería en línea.
- Cítense los conjuntos de entidades y sus claves primarias.

- b. Supóngase que la librería añade casetes de música y discos compactos a su colección. El mismo elemento musical puede estar presente en formato de casete o de disco compacto, con diferentes precios. Extiéndase el diagrama E-R para modelar esta adición, ignorando el efecto sobre las cestas de la compra.
 - c. Extiéndase ahora el diagrama E-R mediante la generalización para modelar el caso en que una cesta de la compra pueda contener cualquier combinación de libros, casetes de música o discos compactos.
- 6.22 En el Apartado 6.5.3 se representó una relación ternaria (que se repite en la Figura 6.29a) mediante relaciones binarias, como se muestra en la Figura 6.29b. Considérese la alternativa mostrada en la Figura 6.29c. Explíquense las ventajas relativas de estas dos representaciones alternativas de una relación ternaria mediante relaciones binarias.
- 6.23 Considérense los esquemas de relación mostrados en el Apartado 6.9.7, que se generaron a partir del diagrama E-R de la Figura 6.25. Para cada esquema, especifíquense las restricciones de clave externa que hay que crear, si es que hay que crear alguna.
- 6.24 Diséñese una jerarquía de especialización–generalización para una compañía de venta de vehículos de motor. La compañía vende motocicletas, coches, furgonetas y autobuses. Justifíquese la ubicación de los atributos en cada nivel de la jerarquía. Explíquese el motivo de que no se deban ubicar en un nivel superior o inferior.
- 6.25 Explíquese la diferencia entre las restricciones definidas por condición y las definidas por el usuario. ¿Cuáles de estas restricciones pueden comprobar el sistema automáticamente? Explíquese la respuesta.
- 6.26 Explíquese la diferencia entre las restricciones disjuntas y las solapadas.
- 6.27 Explíquese la diferencia entre las restricciones totales y las parciales.
- 6.28 Dibújense los equivalentes de UML de los diagramas E-R de las Figuras 6.8c, 6.9, 6.11, 6.12 y 6.20.

Notas bibliográficas

El modelo de datos E-R fue introducido por Chen [1976]. Teorey et al. [1986] presentó una metodología de diseño lógico para las bases de datos relacionales que usa el modelo E-R extendido. La norma de Definición de la integración para el modelado de información (IDEF1X, Integration Definition for Information Modeling) IDEF1X [1993] publicado por el Instituto Nacional de Estados Unidos para Normas y Tecnología (United States National Institute of Standards and Technology, NIST) definió las normas para los diagramas E-R. No obstante, hoy en día se usa gran variedad de notaciones E-R.

Thalheim [2000] proporciona un tratamiento detallado pero propio de libro de texto de la investigación en el modelado E-R. En los libros de texto Batini et al. [1992] y Elmasri y Navathe [2003] se ofrecen explicaciones básicas. Davis et al. [1983] proporciona una colección de artículos sobre el modelo E-R.

En 2004 la versión más reciente de UML era la 1.5, con la versión 2.0 de UML a punto de adoptarse. Véase www.uml.org para obtener más información sobre las normas y las herramientas de UML.

Herramientas

Muchos sistemas de bases de datos ofrecen herramientas para el diseño de bases de datos que soportan los diagramas E-R. Estas herramientas ayudan al diseñador a crear diagramas E-R y pueden crear automáticamente las tablas correspondientes de la base de datos. Véanse las notas bibliográficas del Capítulo 1 para consultar referencias de sitios Web de fabricantes de bases de datos. También hay algunas herramientas de modelado de datos independientes de las bases de datos que soportan los diagramas E-R y los diagramas de clases de UML. Entre ellas están Rational Rose (www.rational.com/products/rose), Microsoft Visio (véase www.microsoft.com/office/visio), ERwin (búsqüese ERwin en el sitio www.cai.com/products), Poseidon para UML (www.gentleware.com) y SmartDraw (www.smartdraw.com).

Diseño de bases de datos relacionales

En este capítulo se considera el problema de diseñar el esquema de una base de datos relacional. Muchos de los problemas que conlleva son parecidos a los de diseño que se han considerado en el Capítulo 6 en relación con el modelo E-R.

En general, el objetivo del diseño de una base de datos relacional es la generación de un conjunto de esquemas de relación que permita almacenar la información sin redundancias innecesarias, pero que también permita recuperarla fácilmente. Esto se consigue mediante el diseño de esquemas que se hallen en la *forma normal* adecuada. Para determinar si el esquema de una relación se halla en una de las formas normales deseables es necesario obtener información sobre la empresa real que se está modelando con la base de datos. Parte de esa información se halla en un diagrama E-R bien diseñado, pero puede ser necesaria información adicional sobre la empresa.

En este capítulo se introduce un enfoque formal al diseño de bases de datos relacionales basado en el concepto de dependencia funcional. Posteriormente se definen las formas normales en términos de las dependencias funcionales y de otros tipos de dependencias de datos. En primer lugar, sin embargo, se examina el problema del diseño relacional desde el punto de vista de los esquemas derivados de un diseño entidad-relación dado.

7.1 Características de los buenos diseños relacionales

El estudio del diseño entidad-relación llevado a cabo en el Capítulo 6 ofrece un excelente punto de partida para el diseño de bases de datos relacionales. Ya se vio en el Apartado 6.9 que es posible generar directamente un conjunto de esquemas de relación a partir del diseño E-R. Evidentemente, la adecuación (o no) del conjunto de esquemas resultante depende, en primer lugar, de la calidad del diseño E-R. Más adelante en este capítulo se estudiarán maneras precisas de evaluar la adecuación de los conjuntos de esquemas de relación. No obstante, es posible llegar a un buen diseño empleando conceptos que ya se han estudiado.

Para facilitar las referencias, en la Figura 7.1 se repiten los esquemas del Apartado 6.9.7.

7.1.1 Alternativa de diseño: esquemas grandes

A continuación se examinan las características de este diseño de base de datos relacional y algunas alternativas. Supóngase que, en lugar de los esquemas *prestatario* y *préstamo*, se considerase el esquema:

$$\text{prestatario_préstamo} = (\text{id_cliente}, \text{número_préstamo}, \text{importe})$$

Esto representa el resultado de la reunión natural de las relaciones correspondientes a *prestatario* y a *préstamo*. Parece una buena idea, ya que es posible expresar algunas consultas empleando menos reuniones, hasta que se considera detenidamente la entidad bancaria que condujo al diseño E-R. Obsérvese

```

sucursal = (nombre_sucursal, ciudad_sucursal, activos)
cliente = (id_cliente, nombre_cliente, calle_cliente, ciudad_cliente)
préstamo = (número_préstamo, importe)
cuenta = (número_cuenta, saldo)
empleado = (id_empleado, nombre_empleado, número_teléfono, fecha_contratación)
nombre_subordinado = (id_empleado, nombre_subordinado)
sucursal_cuenta = (número_cuenta, nombre_sucursal)
sucursal_préstamo = (número_préstamo, nombre_sucursal)
prestatario = (id_cliente, número_préstamo)
impositor = (id_cliente, número_cuenta)
asesor = (id_cliente, id_empleado, tipo)
trabaja_para = (id_empleado_trabajador, id_empleado_jefe)
pago = (número_préstamo, número_pago, fecha_pago, importe_pago)
cuenta_ahorro = (número_cuenta, tasa_interés)
cuenta_corriente = (número_cuenta, importe_descubierto)

```

Figura 7.1 Los esquemas bancarios del Apartado 6.9.7.

que el conjunto de relaciones *prestatario* es varios a varios. Esto permite que cada cliente tenga varios préstamos y, también, que se puedan conceder préstamos conjuntamente a varios clientes. Se tomó esta decisión para que fuese posible representar los préstamos realizados conjuntamente a matrimonios o a consorcios de personas (que pueden participar en una aventura empresarial conjunta). Ese es el motivo de que la clave primaria del esquema *prestatario* consista en *id_cliente* y *número_préstamo* en lugar de estar formada sólo por *número_préstamo*.

Considérese un préstamo que se concede a uno de esos consorcios y considérense las tuplas que deben hallarse en la relación del esquema *prestatario_préstamo*. Supóngase que el préstamo P-100 se concede al consorcio formado por los clientes Santiago (con identificador de cliente 23652), Antonio (con identificador de cliente 15202) y Jorge (con identificador de cliente 23521) por un importe de 10.000 €.

La Figura 7.2 muestra la forma en que es posible representar esta situación empleando *préstamo* y *prestatario* y con el diseño alternativo usando *prestatario_préstamo*. La tupla (P-100, 10.000) de la relación del esquema *préstamo* se reúne con tres tuplas de la relación del esquema *prestatario*, lo que genera tres tuplas de la relación del esquema *prestatario_préstamo*. Obsérvese que en *prestatario_préstamo* es necesario repetir una vez el importe del préstamo por cada cliente integrante del consorcio que solicita el préstamo. Es importante que todas estas tuplas concuerden en lo relativo al importe del préstamo P-100, ya que, en caso contrario, la base de datos quedaría en un estado inconsistente. En el diseño original que utiliza *préstamo* y *prestatario* se almacenaba el importe de cada préstamo exactamente una vez. Esto sugiere que el empleo de *prestatario_préstamo* no es buena idea, ya que se almacena el importe de cada préstamo de manera redundante y se corre el riesgo de que algún usuario actualice el importe del préstamo en una de las tuplas pero no en todas, y, por tanto, genere inconsistencia.

Considérese ahora otra alternativa. Sea *préstamo_importe_sucursal* = (*número_préstamo*, *importe*, *nombre_sucursal*), creada a partir de *préstamo* y de *sucursal_préstamo* (mediante una reunión de las relaciones correspondientes). Esto se parece al ejemplo que se acaba de considerar, pero con una importante diferencia. En este caso, *número_préstamo* es la clave primaria de los dos esquemas, *sucursal_préstamo* y *préstamo* y, por tanto, también lo es de *préstamo_importe_sucursal*. Esto se debe a que el conjunto de relaciones *sucursal_préstamo* está definido como varios a uno, a diferencia del conjunto de relaciones *prestatario* del ejemplo anterior. Para cada préstamo sólo hay una sucursal asociada. Por tanto, cada número de préstamo concreto sólo aparece una vez en *sucursal_préstamo*. Supóngase que el préstamo P-100 está asociado con la sucursal de Soria. La Figura 7.3 muestra la manera en que esto se representa empleando *préstamo_importe_sucursal*. La tupla (P-100, 10.000) de la relación del esquema *préstamo* se reúne con una sola tupla de la relación del esquema *sucursal_préstamo*, lo que sólo genera una tupla de la relación del esquema *préstamo_importe_sucursal*. No hay repetición de información en *préstamo_importe_sucursal* y, por tanto, se evitan los problemas que se encontraron en el ejemplo anterior.

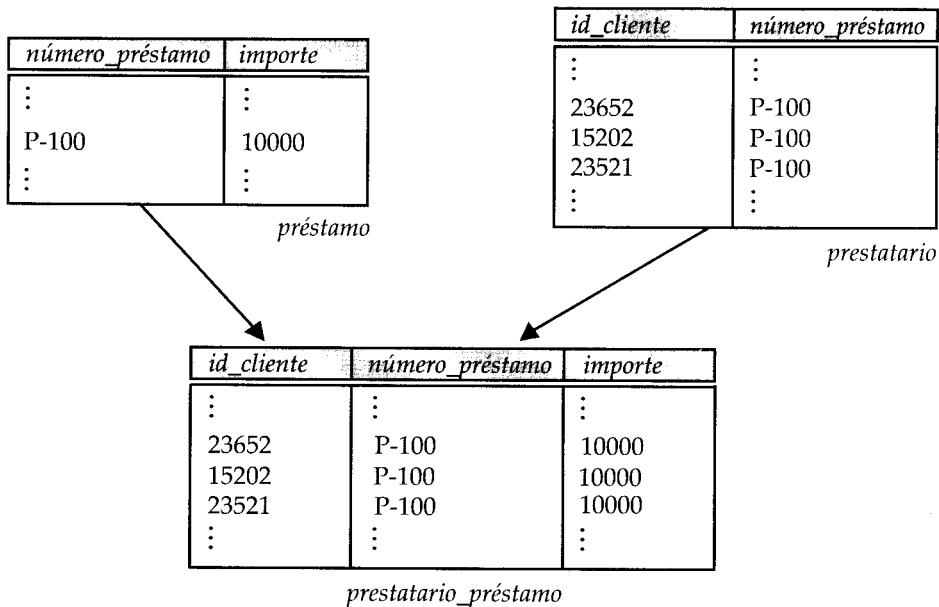


Figura 7.2 Lista parcial de las tuplas de las relaciones *préstamo*, *prestatario* y *prestatario_préstamo*.

Antes de acceder finalmente al empleo de *préstamo_importe_sucursal* en lugar de *sucursal_préstamo* y de *préstamo*, hay otro aspecto que se debe considerar. Puede que se desee registrar en la base de datos un préstamo y su sucursal asociada antes de que se haya determinado su importe. En el diseño antiguo, el esquema *sucursal_préstamo* puede hacerlo pero, con el diseño revisado *préstamo_importe_sucursal*, habría que crear una tupla con un valor nulo para *importe*. En algunos casos los valores nulos pueden crear problemas, como se vio en el estudio de SQL. No obstante, si se decide que en este caso no suponen ningún problema, se puede aceptar el empleo del diseño revisado.

Los dos ejemplos que se acaban de examinar muestran la importancia de la naturaleza de las claves primarias al decidir si las combinaciones de esquemas tienen sentido. Se han presentado problemas —concretamente, la repetición de información— cuando el atributo de reunión (*número_préstamo*) no era la clave primaria de *los dos* esquemas que se combinaban.

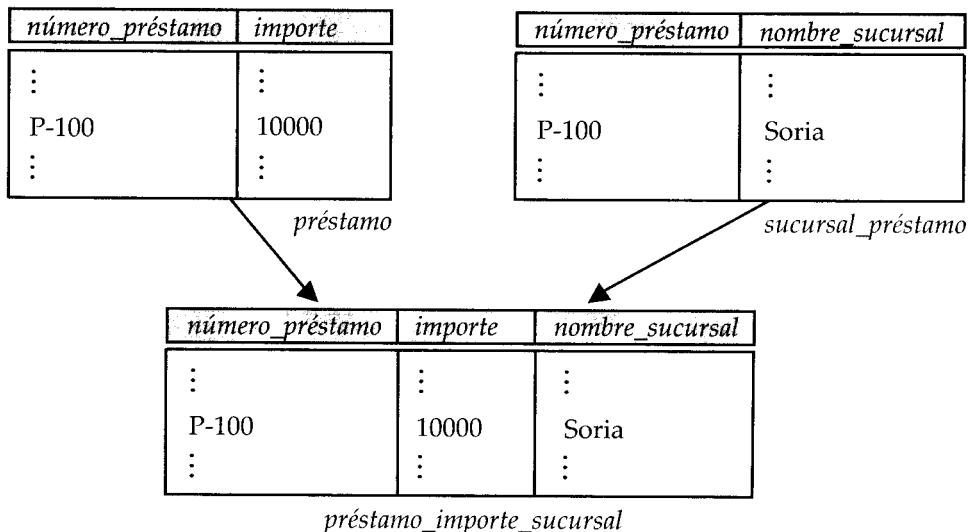


Figura 7.3 Lista parcial de las tuplas de las relaciones *préstamo*, *sucursal_préstamo* y *préstamo_importe_sucursal*.

7.1.2 Alternativa de diseño: esquemas pequeños

Supóngase nuevamente que, de algún modo, se ha comenzado a trabajar con el esquema *prestatario_préstamo*. ¿Cómo se reconoce que necesita que haya repetición de la información y que hay que dividirlo en dos esquemas, *prestatario* y *préstamo*? Como no se dispone de los esquemas *prestatario* y *préstamo*, se carece de la información sobre la clave primaria que se empleó para describir el problema de *prestatario_préstamo*.

Mediante la observación del contenido de las relaciones reales del esquema *prestatario_préstamo* se puede percibir la repetición de información resultante de tener que relacionar el importe del préstamo una vez por cada prestatario asociado con cada uno de ellos. Sin embargo, se trata de un proceso en el que no se puede confiar. Las bases de datos reales tienen gran número de esquemas y un número todavía mayor de atributos. El número de tuplas puede ser del orden de millones. Descubrir la repetición puede resultar costoso. Hay un problema más fundamental todavía en este enfoque. No permite determinar si la carencia de repeticiones es meramente un caso especial “afortunado” o la manifestación de una regla general. En el ejemplo anterior, ¿cómo se sabe que en la entidad bancaria cada préstamo (identificado por su número correspondiente) sólo debe tener un importe? ¿El hecho de que el préstamo P-100 aparezca tres veces con el mismo importe es sólo una coincidencia? Estas preguntas no se pueden responder sin volver a la propia empresa y comprender sus reglas de funcionamiento. En concreto, hay que averiguar si el banco exige que cada préstamo (identificado por su número correspondiente) sólo tenga un importe.

En el caso de *prestatario_préstamo*, el proceso de creación del diseño E-R logró evitar la creación de ese esquema. Sin embargo, esta situación fortuita no se produce siempre. Por tanto, hay que permitir que el diseñador de la base de datos especifique normas como “cada valor de *número_préstamo* corresponde, como máximo, a un *importe*”, incluso en los casos en los que *número_préstamo* no sea la clave primaria del esquema en cuestión. En otras palabras, hay que escribir una norma que diga “si hay un esquema (*número_préstamo, importe*), entonces *número_préstamo* puede hacer de clave primaria”. Esta regla se especifica como la **dependencia funcional** $número_{préstamo} \rightarrow importe$. Dada esa regla, ya se tiene suficiente información para reconocer el problema del esquema *prestatario_préstamo*. Como *número_préstamo* no puede ser la clave primaria de *prestatario_préstamo* (porque puede que cada préstamo necesite varias tuplas de la relación del esquema *prestatario_préstamo*), tal vez haya que repetir el importe del préstamo.

Observaciones como éstas y las reglas (especialmente las dependencias funcionales) que generan permiten al diseñador de la base de datos reconocer las situaciones en que hay que dividir, o descomponer, un esquema en dos o más. No es difícil comprender que la manera correcta de descomponer *prestatario_préstamo* es dividirlo en los esquemas *prestatario* y *préstamo*, como en el diseño original. Hallar la descomposición correcta es mucho más difícil para esquemas con gran número de atributos y varias dependencias funcionales. Para trabajar con ellos hay que confiar en una metodología formal que se desarrolle más avanzado este capítulo.

No todas las descomposiciones de los esquemas resultan útiles. Considérese el caso extremo en que todo lo que tengamos sean esquemas con un solo atributo. No se puede expresar ninguna relación interesante de ningún tipo. Considérese ahora un caso menos extremo en el que se decida descomponer el esquema *empleado* en

$$\begin{aligned} empleado1 &= (\underline{id_empleado}, nombre_empleado) \\ empleado2 &= (nombre_empleado, número_teléfono, fecha_contratación) \end{aligned}$$

El problema con esta descomposición surge de la posibilidad de que la empresa tenga dos empleados que se llamen igual. Esto no es improbable en la práctica, ya que muchas culturas tienen algunos nombres muy populares y, además, puede que los niños se llamen como sus padres. Por supuesto, cada persona tiene un identificador de empleado único, que es el motivo de que se pueda utilizar *id_empleado* como clave primaria. A modo de ejemplo, supóngase que dos empleados llamados Koldo trabajan para el mismo banco y tienen las tuplas siguientes de la relación del esquema *empleado* del diseño original:

(1234567890, Koldo, 918820000, 29/03/1984)
(9876543210, Koldo, 918699999, 16/01/1981)

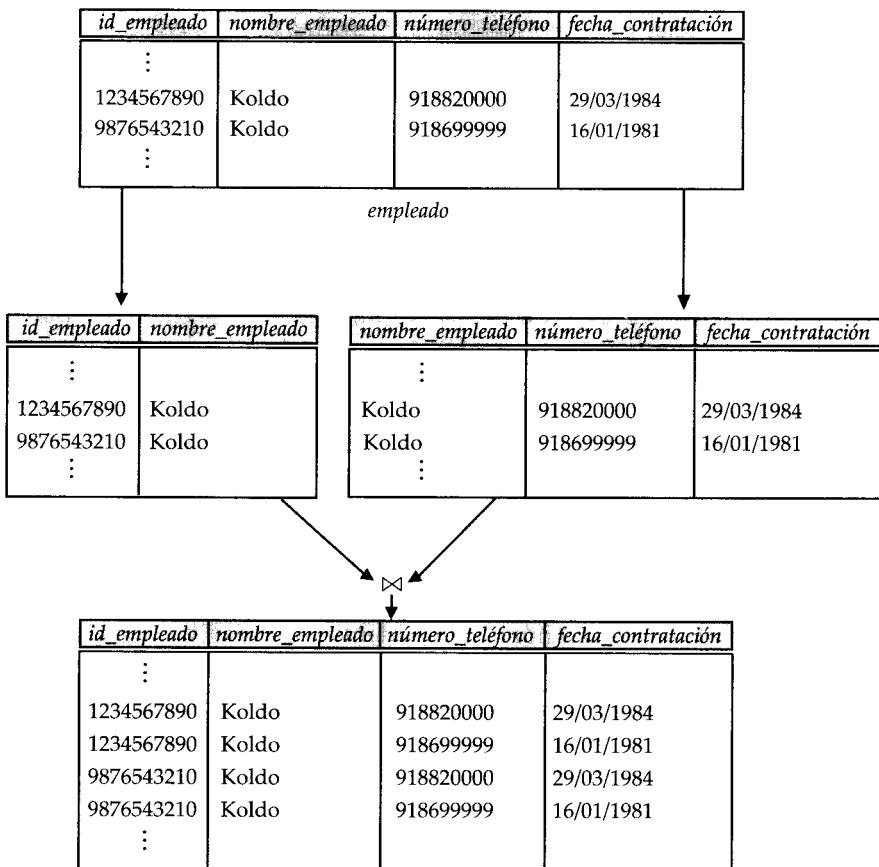


Figura 7.4 Pérdida de información debida a una mala descomposición.

La Figura 7.4 muestra estas tuplas, las tuplas resultantes al utilizar los esquemas procedentes de la descomposición y el resultado que se obtendría si se intentara volver a generar las tuplas originales mediante una reunión natural. Como se ve en la figura, las dos tuplas originales aparecen en el resultado junto con dos tuplas nuevas que mezclan de manera incorrecta valores de fechas correspondientes a los dos empleados llamados Koldo. Aunque se dispone de más tuplas, se tiene en realidad menos información en el sentido siguiente: se puede indicar que un determinado número de teléfono y una fecha de contratación dada corresponden a alguien llamado Koldo, pero no se puede distinguir a cuál de ellos. Por tanto, la descomposición propuesta es incapaz de representar algunos datos importantes de la entidad bancaria. Evidentemente, es preferible evitar este tipo de descomposiciones. Estas descomposiciones se denominan **descomposiciones con pérdidas**, y, a la inversa, aquéllas que no tienen pérdidas se denominan **descomposiciones sin pérdidas**.

7.2 Dominios atómicos y la primera forma normal

El modelo E-R permite que los conjuntos de entidades y los de relaciones tengan atributos con algún tipo de subestructura. Concretamente, permite atributos multivalorados, como *nombre_subordinado* de la Figura 6.25, y atributos compuestos (como el atributo *dirección* con los atributos componentes *calle* y *ciudad*). Cuando se crean tablas a partir de los diseños E-R que contienen ese tipo de atributos, se elimina esa subestructura. Para los atributos compuestos, se deja que cada atributo componente sea un atributo de pleno derecho. Para los atributos multivalorados, se crea una tupla por cada elemento del conjunto multivalorado.

En el modelo relacional se formaliza esta idea de que los atributos no tienen subestructuras. Un dominio es **atómico** si se considera que los elementos de ese dominio son unidades indivisibles. Se dice que

el esquema de la relación R está en la **primera forma normal** (1FN) si los dominios de todos los atributos de R son atómicos.

Los conjuntos de nombres son ejemplos de valores no atómicos. Por ejemplo, si el esquema de la relación *empleado* incluyera el atributo *hijos*, los elementos de cuyo dominio son conjuntos de nombres, el esquema no estaría en la primera forma normal.

Los atributos compuestos, como el atributo *dirección* con los atributos componentes *calle* y *ciudad*, tienen también dominios no atómicos.

Se da por supuesto que los números enteros son atómicos, por lo que el conjunto de los números enteros es un dominio atómico; el conjunto de todos los conjuntos de números enteros es un dominio no atómico. La diferencia estriba en que normalmente no se considera que los enteros tengan subpartes, pero sí se considera que las tienen los conjuntos de enteros—es decir, los enteros que componen el conjunto. Pero lo importante no es lo que sea el propio dominio, sino el modo en que se utilizan los elementos de ese dominio en la base de datos. El dominio de todos los enteros sería no atómico si se considerara que cada entero es una lista ordenada de cifras.

Como ilustración práctica del punto anterior, considérese una organización que asigna a los empleados números de identificación de la manera siguiente: las dos primeras letras especifican el departamento y las cuatro cifras restantes son un número único para cada empleado dentro de ese departamento. Ejemplos de estos números pueden ser *CS0012* y *EE1127*. Estos números de identificación pueden dividirse en unidades menores y, por tanto, no son atómicos. Si el esquema de la relación tuviera un atributo cuyo dominio consistiera en números de identificación así codificados, el esquema no se hallaría en la primera forma normal.

Cuando se utilizan números de identificación de este tipo, se puede averiguar el departamento de cada empleado escribiendo código que analice la estructura de los números de identificación. Ello exige programación adicional, y la información queda codificada en el programa de aplicación en vez de en la base de datos. Surgen nuevos problemas si se utilizan esos números de identificación como claves primarias: cada vez que un empleado cambie de departamento, habrá que modificar su número de identificación en todos los lugares en que aparezca, lo que puede constituir una tarea difícil; en caso contrario, el código que interpreta ese número dará un resultado erróneo.

El empleo de atributos con el valor definido por un conjunto puede llevar a diseños con almacenamiento de datos redundante; lo que, a su vez, puede dar lugar a inconsistencias. Por ejemplo, en lugar de representar la relación entre las cuentas y los clientes como la relación independiente *impositor*, puede que el diseñador de bases de datos esté tentado a almacenar un conjunto de *titulares* con cada cuenta, y un conjunto de *cuentas* con cada cliente. Siempre que se cree una cuenta, o se actualice el conjunto de titulares de una cuenta, habrá que llevar a cabo la actualización en dos lugares; no llevar a cabo esas dos actualizaciones puede dejar la base de datos en un estado inconsistente. Guardar sólo uno de esos conjuntos evitaría la información repetida, pero complicaría algunas consultas.

Algunos tipos de valores no atómicos pueden resultar útiles, aunque deben utilizarse con cuidado. Por ejemplo, los atributos con valores compuestos suelen resultar útiles, y los atributos con el valor determinado por un conjunto también resultan útiles en muchos casos, que es el motivo por el que el modelo E-R los soporta. En muchos dominios en los que las entidades tienen una estructura compleja, la imposición de la representación en la primera forma normal supone una carga innecesaria para el programador de las aplicaciones, que tiene que escribir código para convertir los datos a su forma atómica. También hay sobrecarga en tiempo de ejecución por la conversión de los datos entre su forma habitual y su forma atómica. Por tanto, el soporte de los valores no atómicos puede resultar muy útil en ese tipo de dominios. De hecho, los sistemas modernos de bases de datos soportan muchos tipos de valores no atómicos, como se verá en el Capítulo 9. Sin embargo, en este capítulo nos limitamos a las relaciones en la primera forma normal y, por tanto, todos los dominios son atómicos.

7.3 Descomposición mediante dependencias funcionales

En el Apartado 7.1 se indicó que hay una metodología formal para evaluar si un esquema relacional debe descomponerse. Esta metodología se basa en los conceptos de clave y de dependencia funcional.

7.3.1 Claves y dependencias funcionales

Las claves y, más en general, las dependencias funcionales son restricciones de la base de datos que exigen que las relaciones cumplan determinadas propiedades. Las relaciones que cumplen todas esas restricciones son **legales**.

En el Capítulo 6 se definió el concepto de *superclave* de la manera siguiente. Sea R el esquema de una relación. El subconjunto C de R es una **superclave** de R si, en cualquier relación legal $r(R)$, para todos los pares t_1 y t_2 de tuplas de r tales que $t_1 \neq t_2$, $t_1[C] \neq t_2[C]$. Es decir, ningún par de tuplas de una relación legal $r(R)$ puede tener el mismo valor del conjunto de atributos C .

Mientras que una clave es un conjunto de atributos que identifica de manera única toda una tupla, una dependencia funcional permite expresar restricciones que identifican de manera única el valor de determinados atributos. Considérese el esquema de una relación R y sean $\alpha \subseteq R$ y $\beta \subseteq R$. La **dependencia funcional** $\alpha \rightarrow \beta$ se cumple para el esquema R si, en cualquier relación legal $r(R)$, para todos los pares de tuplas t_1 y t_2 de r tales que $t_1[\alpha] = t_2[\alpha]$, también se cumple que $t_1[\beta] = t_2[\beta]$.

Empleando la notación para la dependencia funcional, se dice que C es superclave de R si $C \rightarrow R$. Es decir, C es superclave si, siempre que $t_1[C] = t_2[C]$, también se cumple que $t_1[R] = t_2[R]$ (es decir, $t_1 = t_2$).

Las dependencias funcionales permiten expresar las restricciones que no se pueden expresar con superclaves. En el Apartado 7.1.2 se consideró el esquema

$$\text{prestatario_préstamo} = (\underline{\text{id_cliente}}, \underline{\text{número_préstamo}}, \text{importe})$$

en el que se cumple la dependencia funcional $\text{número_préstamo} \rightarrow \text{importe}$, ya que por cada préstamo (identificado por número_préstamo) hay un solo importe. El hecho de que el par de atributos $(\text{id_cliente}, \text{número_préstamo})$ forma una superclave para $\text{prestatario_préstamo}$ se denota escribiendo

$$\text{id_cliente}, \text{número_préstamo} \rightarrow \text{id_cliente}, \text{número_préstamo}, \text{importe}$$

o, de manera equivalente,

$$\text{id_cliente}, \text{número_préstamo} \rightarrow \text{prestatario_préstamo}$$

Las dependencias funcionales se emplean de dos maneras:

1. Para probar las relaciones y ver si son legales de acuerdo con un conjunto dado de dependencias funcionales. Si una relación r es legal según el conjunto F de dependencias funcionales, se dice que r **satisface** F .
2. Para especificar las restricciones del conjunto de relaciones legales. Así, sólo habrá que preocuparse de las relaciones que satisfagan un conjunto dado de dependencias funcionales. Si uno desea restringirse a las relaciones del esquema R que satisfacen el conjunto F de dependencias funcionales, se dice que F se **cumple** en R .

Considérese la relación r de la Figura 7.5, para ver las dependencias funcionales que se satisfacen. Obsérvese que se satisface $A \rightarrow C$. Hay dos tuplas que tienen un valor para A de a_1 . Estas tuplas tienen el mismo valor de C —por ejemplo, c_1 . De manera parecida, las dos tuplas con un valor para A de a_2

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4

Figura 7.5 Relación de ejemplo r .

tienen el mismo valor de C , c_2 . No hay más pares de tuplas diferentes que tengan el mismo valor de A . Sin embargo, la dependencia funcional $C \rightarrow A$ no se satisface. Para verlo, considérense las tuplas $t_1 = (a_2, b_3, c_2, d_3)$ y $t_2 = (a_3, b_3, c_2, d_4)$. Estas dos tuplas tienen el mismo valor de C , c_2 , pero tienen valores diferentes para A , a_2 y a_3 , respectivamente. Por tanto, se ha hallado un par de tuplas t_1 y t_2 tales que $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.

Se dice que algunas dependencias funcionales son **triviales** porque las satisfacen todas las relaciones. Por ejemplo, $A \rightarrow A$ la satisfacen todas las relaciones que implican al atributo A . La lectura literal de la definición de dependencia funcional deja ver que, para todas las tuplas t_1 y t_2 tales que $t_1[A] = t_2[A]$, se cumple que $t_1[A] = t_2[A]$. De manera análoga, $AB \rightarrow A$ la satisfacen todas las relaciones que implican al atributo A . En general, las dependencias funcionales de la forma $\alpha \rightarrow \beta$ son **triviales** si se cumple la condición $\beta \subseteq \alpha$.

Es importante darse cuenta de que una relación dada puede, en cualquier momento, satisfacer algunas dependencias funcionales cuyo cumplimiento no sea necesario en el esquema de la relación. En la relación *cliente* de la Figura 2.4 puede verse que se satisface *calle_cliente* \rightarrow *ciudad_cliente*. Sin embargo, se sabe que en el mundo real dos ciudades diferentes pueden tener calles que se llamen igual. Por tanto, es posible, en un momento dado, tener un ejemplar de la relación *cliente* en el que no se satisfaga *calle_cliente* \rightarrow *ciudad_cliente*. Por consiguiente, no se incluirá *calle_cliente* \rightarrow *ciudad_cliente* en el conjunto de dependencias funcionales que se cumplen en la relación *cliente*.

Dado un conjunto de dependencias funcionales F que se cumple en una relación r , es posible que se pueda inferir que también se deban cumplir en esa misma relación otras dependencias funcionales. Por ejemplo, dado el esquema $r = (A, B, C)$, si se cumplen en r las dependencias funcionales $A \rightarrow B$ y $B \rightarrow C$, se puede inferir que también $A \rightarrow C$ se debe cumplir en r . Ya que, dado cualquier valor de A , sólo puede haber un valor correspondiente de B y, para ese valor de B , sólo puede haber un valor correspondiente de C . Más adelante, en el Apartado 7.4.1, se estudia la manera de realizar esas inferencias.

Se utiliza la notación F^+ para denotar el **cierre** del conjunto F , es decir, el conjunto de todas las dependencias funcionales que pueden inferirse dado el conjunto F . Evidentemente, F^+ es un superconjunto de F .

7.3.2 Forma normal de Boyce-Codd

Una de las formas normales más deseables que se pueden obtener es la **forma normal de Boyce-Codd (FNBC)**. Elimina todas las redundancias que se pueden descubrir a partir de las dependencias funcionales aunque, como se verá en el Apartado 7.6, puede que queden otros tipos de redundancia. El esquema de relación R está en la FNBC respecto al conjunto F de dependencias funcionales si, para todas las dependencias funcionales de F^+ de la forma $\alpha \rightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple, al menos, una de las siguientes condiciones:

- $\alpha \rightarrow \beta$ es una dependencia funcional trivial (es decir, $\beta \subseteq \alpha$).
- α es superclave del esquema R .

Los diseños de bases de datos están en la FNBC si cada miembro del conjunto de esquemas de relación que constituye el diseño se halla en la FNBC.

Ya se ha visto que el esquema *prestatario_préstamo* = (*id_cliente*, *número_préstamo*, *importe*) no se halla en FNBC. La dependencia funcional *número_préstamo* \rightarrow *importe* se cumple en *prestatario_préstamo*, pero *número_préstamo* no es superclave (ya que, como se recordará, los préstamos se pueden conceder a consorcios formados por varios clientes). En el Apartado 7.1.2 se vio que la descomposición de *prestatario_préstamo* en *prestatario* y *préstamo* es mejor diseño. El esquema *prestatario* se halla en la FNBC, ya que no se cumple en él ninguna dependencia funcional que no sea trivial. El esquema *préstamo* tiene una dependencia funcional no trivial que se cumple, *número_préstamo* \rightarrow *importe*, pero *número_préstamo* es superclave (realmente, en este caso, la clave primaria) de *préstamo*. Por tanto, *préstamo* se halla en la FNBC.

Ahora se va a definir una regla general para la descomposición de esquemas que no se hallen en la FNBC. Sea R un esquema que no se halla en la FNBC. En ese caso, queda, al menos, una dependencia funcional no trivial $\alpha \rightarrow \beta$ tal que α no es superclave de R . En el diseño se sustituye R por dos esquemas:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

En el caso anterior de *prestatario_préstamo*, $\alpha = \text{número_préstamo}$, $\beta = \text{importe}$ y *prestatario_préstamo* se sustituye por

- $(\alpha \cup \beta) = (\text{número_préstamo}, \text{importe})$
- $(R - (\beta - \alpha)) = (\text{id_cliente}, \text{número_préstamo})$

En este caso resulta que $\beta - \alpha = \beta$. Hay que definir la regla tal y como se ha hecho para que trate correctamente las dependencias funcionales que tienen atributos que aparecen a los dos lados de la flecha. Las razones técnicas para esto se tratarán más adelante, en el Apartado 7.5.1.

Cuando se descomponen esquemas que no se hallan en la FNBC, puede que uno o varios de los esquemas resultantes no se hallen en la FNBC. En ese caso, hacen falta más descomposiciones, cuyo resultado final es un conjunto de esquemas FNBC.

7.3.3 FNBC y la conservación de las dependencias

Se han visto varias maneras de expresar las restricciones de consistencia de las bases de datos: restricciones de clave primaria, dependencias funcionales, restricciones *check*, asertos y disparadores. La comprobación de estas restricciones cada vez que se actualiza la base de datos puede ser costosa y, por tanto, resulta útil diseñar la base de datos de manera que las restricciones se puedan comprobar de manera eficiente. En concreto, si la comprobación de las dependencias funcionales puede realizarse considerando sólo una relación, el coste de comprobación de esa restricción será bajo. Se verá que la descomposición en la FNBC puede impedir la comprobación eficiente de determinadas dependencias funcionales.

Para ilustrar esto, supóngase que se lleva a cabo una modificación aparentemente pequeña en el modo en que opera la entidad bancaria. En el diseño de la Figura 6.25 cada cliente sólo puede tener un empleado como “asesor personal”. Esto se deduce de que el conjunto de relaciones *asesor* es varios a uno de *cliente* a *empleado*. La “pequeña” modificación que se va a llevar a cabo es que cada cliente pueda tener más de un asesor personal pero, como máximo, uno por sucursal.

Esto se puede implementar en el diagrama E-R haciendo que el conjunto de relaciones *asesor* sea varios a varios (ya que ahora cada cliente puede tener más de un asesor personal) y añadiendo un nuevo conjunto de relaciones, *trabaja_en*, entre *empleado* y *sucursal*, que indique los pares empleado-sucursal, en los que el empleado trabaja en la sucursal. Hay que hacer *trabaja_en* varios a uno de *empleado* a *sucursal*, ya que cada sucursal puede tener varios empleados, pero cada empleado sólo puede trabajar en una sucursal. La Figura 7.6 muestra un subconjunto de la Figura 6.25 con estas adiciones.

No obstante, hay un fallo en este diseño. Permite que cada cliente tenga dos (o más) asesores personales que trabajen en la misma sucursal, algo que el banco no permite. Lo ideal sería que hubiera un solo conjunto de relaciones al que se pudiera hacer referencia para hacer que se cumpla esta restricción. Esto exige que se considere una manera diferente de modificar el diseño E-R. En lugar de añadir el conjunto de relaciones *trabaja_en*, se sustituye el conjunto de relaciones *asesor* por la relación ternaria *sucursal_asesor*, que implica a los conjuntos de entidades *cliente*, *empleado* y *sucursal* y es varios a uno del par *cliente*, *empleado* a *sucursal*, como puede verse en la Figura 7.7. Como este diseño permite que un solo conjunto de relaciones represente la restricción, supone una ventaja significativa respecto del primer enfoque considerado.

Sin embargo, la diferencia entre estos dos enfoques no es tan clara. El esquema derivado de *sucursal_asesor* es

$$\text{sucursal_asesor} = (\text{id_cliente}, \text{id_empleado}, \text{nombre_sucursal}, \text{tipo})$$

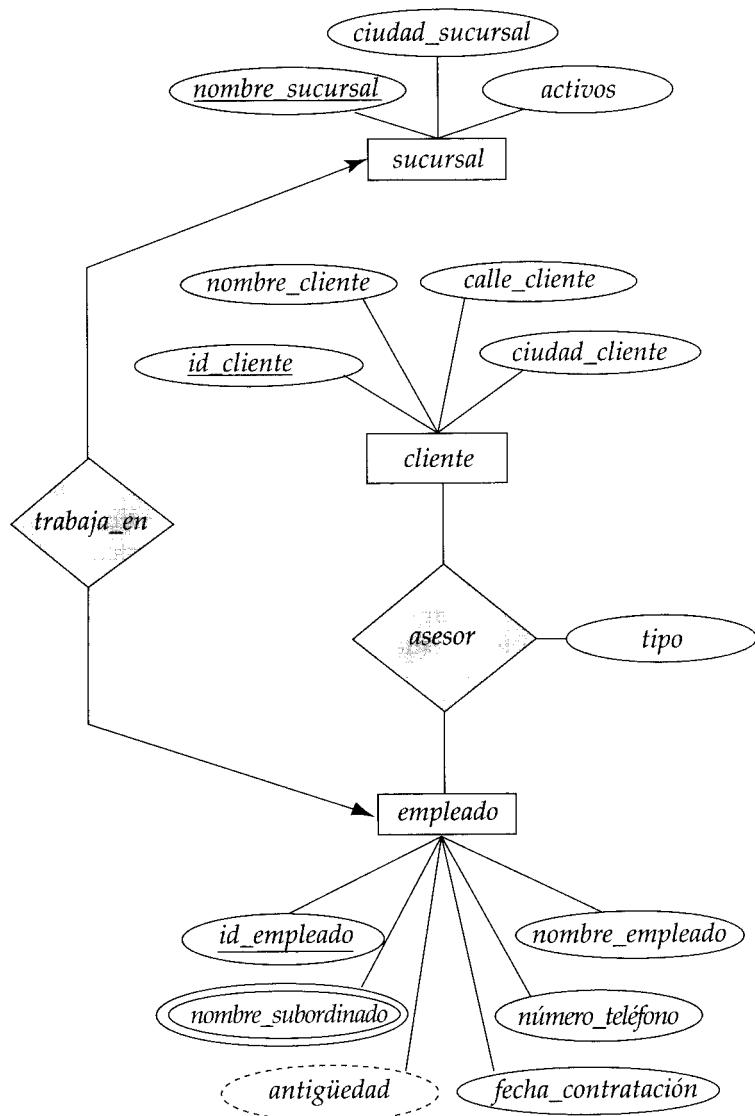


Figura 7.6 Los conjuntos de relaciones *asesor* y *trabaja_en*.

Como cada empleado sólo puede trabajar en una sucursal, en la relación del esquema *sucursal_asesor* sólo puede haber un valor de *nombre_sucursal* asociado con cada valor de *id_empleado*; es decir:

$$id_empleado \rightarrow nombre_sucursal$$

No obstante, no queda más remedio que repetir el nombre de la sucursal cada vez que un empleado participa en la relación *sucursal_asesor*. Es evidente que *sucursal_asesor* no se halla en la FNBC, ya que *id_empleado* no es superclave. De acuerdo con la regla para la descomposición en la FNBC, se obtiene:

$$\begin{array}{l} (id_cliente, id_empleado, tipo) \\ \hline (id_empleado, nombre_sucursal) \end{array}$$

Este diseño, que es exactamente igual que el primer enfoque que utilizaba el conjunto de relaciones *trabaja_en*, dificulta que se haga cumplir la restricción de que cada cliente pueda tener, como máximo, un asesor en cada sucursal. Esta restricción se puede expresar mediante la dependencia funcional

$$id_cliente, nombre_sucursal \rightarrow id_empleado$$

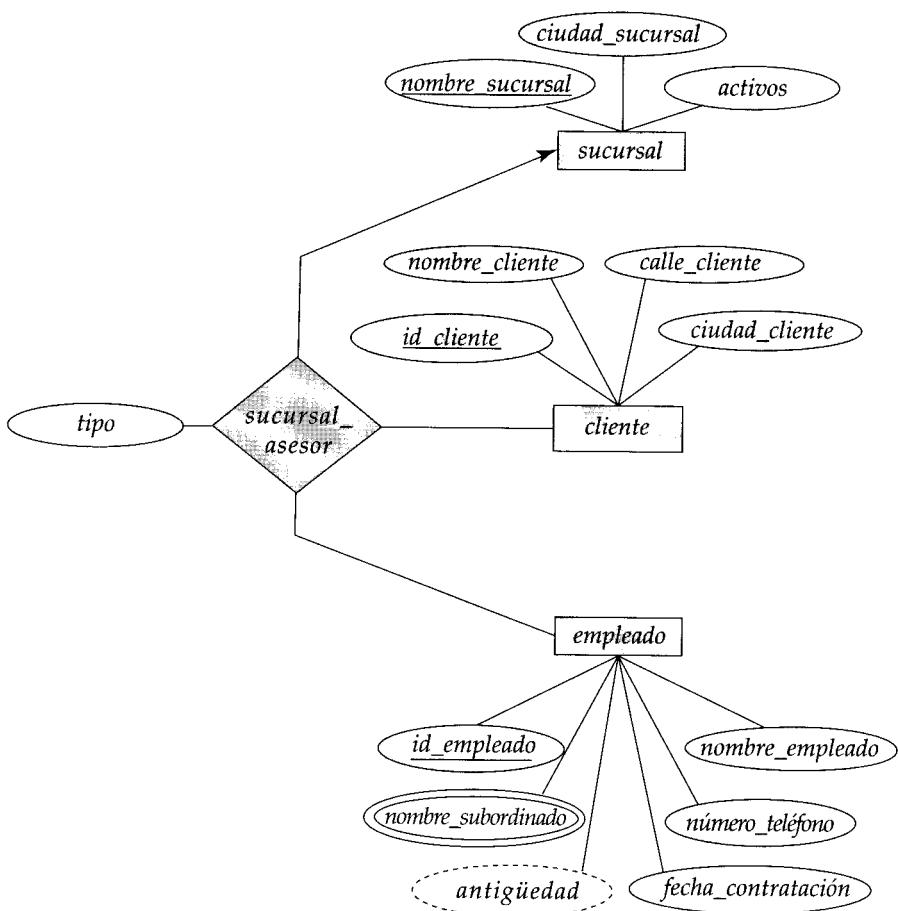


Figura 7.7 El conjunto de relaciones *sucursal_asesor*.

y hacer notar que en el diseño FNBC escogido no hay ningún esquema que incluya todos los atributos que aparecen en esta dependencia funcional. Como el diseño hace computacionalmente difícil que se haga cumplir esta dependencia funcional, se dice que el diseño no **conserva las dependencias**¹. Como la conservación de las dependencias suele considerarse deseable, se considera otra forma normal, más débil que la FNBC, que permite que se conserven las dependencias. Esa forma normal se denomina tercera forma normal².

7.3.4 Tercera forma normal

La FNBC exige que todas las dependencias no triviales sean de la forma $\alpha \rightarrow \beta$, donde α es una superclave. La tercera forma normal (3NF) relaja ligeramente esta restricción al permitir dependencias funcionales no triviales cuya parte izquierda no sea una superclave. Antes de definir la 3NF hay que recordar que las claves candidatas son superclaves mínimas—es decir, superclaves de las que ningún subconjunto propio sea también superclave.

El esquema de relación R está en **tercera forma normal** respecto a un conjunto F de dependencias funcionales si, para todas las dependencias funcionales de F^+ de la forma $\alpha \rightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple, al menos, una de las siguientes condiciones:

1. Técnicamente es posible que una dependencia cuyos atributos no aparezcan en su totalidad en ningún esquema se siga haciendo cumplir de manera implícita, debido a la presencia de otras dependencias que la impliquen lógicamente. Este caso se abordará más adelante, en el Apartado 7.4.5.
2. Puede que se haya observado que se ha omitido la segunda forma normal. Sólo tiene significación histórica y, en la práctica, no se utiliza.

- $\alpha \rightarrow \beta$ es una dependencia funcional trivial.
- α es superclave de R .
- Cada atributo A de $\beta - \alpha$ está contenido en alguna clave candidata de R .

Obsérvese que la tercera condición no dice que una sola clave candidata deba contener todos los atributos de $\beta - \alpha$; cada atributo A de $\beta - \alpha$ puede estar contenido en una clave candidata *diferente*.

Las dos primeras alternativas son iguales que las dos alternativas de la definición de la FNBC. La tercera alternativa de la definición de la 3FN parece bastante poco intuitiva, y no resulta evidente el motivo de su utilidad. Representa, en cierto sentido, una relajación mínima de las condiciones de la FNBC que ayuda a garantizar que cada esquema tenga una descomposición que conserve las dependencias en la 3FN. Su finalidad se aclarará más adelante, cuando se estudie la descomposición en la 3FN.

Obsérvese que cualquier esquema que satisfaga la FNBC satisface también la 3FN, ya que cada una de sus dependencias funcionales satisfará una de las dos primeras alternativas. Por tanto, la FNBC es una forma normal más restrictiva que la 3FN.

La definición de la 3FN permite ciertas dependencias funcionales que no se permiten en la FNBC. Las dependencias $\alpha \rightarrow \beta$ que sólo satisfacen la tercera alternativa de la definición de la 3FN no se permiten en la FNBC, pero sí en la 3FN³.

Considérese nuevamente el ejemplo *sucursal_asesor* y la dependencia funcional

$$id_empleado \rightarrow nombre_sucursal$$

que hacía que el esquema no se hallara en FNBC. Obsérvese que, en este caso, $\alpha = id_empleado$, $\beta = nombre_sucursal$ y $\beta - \alpha = nombre_sucursal$. Resulta que *nombre_sucursal* está contenido en una clave candidata y que, por tanto, *sucursal_asesor* se halla en la 3FN. Probar esto, no obstante, exige un pequeño esfuerzo.

Se sabe que, además de las dependencias funcionales

$$\begin{aligned} id_empleado &\rightarrow nombre_sucursal \\ id_cliente, nombre_sucursal &\rightarrow id_empleado \end{aligned}$$

que se cumplen, la dependencia funcional

$$id_cliente, id_empleado \rightarrow sucursal_asesor$$

se cumple como consecuencia de que $(id_cliente, id_empleado)$ es la clave primaria. Esto hace de $(id_cliente, id_empleado)$ clave candidata. Por supuesto, no contiene *nombre_sucursal*, por lo que hay que ver si hay otras claves candidatas. Se concluye que el conjunto de atributos $(id_cliente, nombre_sucursal)$ es clave candidata. Veamos el motivo.

Dados unos valores concretos de *id_cliente* y de *nombre_sucursal*, se sabe que sólo hay un valor asociado de *id_empleado*, ya que

$$id_cliente, nombre_sucursal \rightarrow id_empleado$$

Pero entonces, para esos valores concretos de *id_cliente* y de *id_empleado*, sólo puede haber una tupla asociada de *sucursal_asesor*, ya que

$$id_cliente, id_empleado \rightarrow sucursal_asesor$$

Por tanto, se ha argumentado que $(id_cliente, nombre_sucursal)$ es superclave. Como resulta que ni *id_cliente* ni *nombre_sucursal* por separado son superclaves, $(id_cliente, nombre_sucursal)$ es clave candidata.

3. Estas dependencias son ejemplos de **dependencias transitivas** (véase el Ejercicio práctico 7.14). La definición original de la 3NF se hizo en términos de las dependencias transitivas. La definición que aquí se utiliza es equivalente, pero más fácil de comprender.

ta. Dado que esta clave candidata contiene *nombre_sucursal*, la dependencia funcional

$$id_empleado \rightarrow nombre_sucursal$$

no viola las reglas de la 3FN.

La demostración de que *sucursal_asesor* se halla en la 3NF ha supuesto cierto esfuerzo. Por esta razón (y por otras), resulta útil adoptar un enfoque estructurado y formal del razonamiento sobre las dependencias funcionales, las formas normales y la descomposición de los esquemas, lo cual se hará en el Apartado 7.4.

Se ha visto el equilibrio que hay que guardar entre la FNBC y la 3NF cuando no hay ningún diseño FNBC que conserve las dependencias. Estos equilibrios se describen con más detalle en el Apartado 7.5.3; en ese apartado también se describe un enfoque de la comprobación de dependencias mediante vistas materializadas que permite obtener las ventajas de la FNBC y de la 3NF.

7.3.5 Formas normales superiores

Puede que en algunos casos el empleo de las dependencias funcionales para la descomposición de los esquemas no sea suficiente para evitar la repetición innecesaria de información. Considérese una ligera variación de la definición del conjunto de entidades *empleado* en la que se permita que los empleados tengan varios números de teléfono, alguno de los cuales puede ser compartido entre varios empleados. Entonces, *número_teléfono* será un atributo multivalorado y, de acuerdo con las reglas para la generación de esquemas a partir de los diseños E-R, habrá dos esquemas, uno por cada uno de los atributos multivalorados *número_teléfono* y *nombre_subordinado*:

$$(id_empleado, nombre_subordinado) (id_empleado, número_teléfono)$$

Si se combinan estos esquemas para obtener

$$(id_empleado, nombre_subordinado, número_teléfono)$$

se descubre que el resultado se halla en la FNBC, ya que sólo se cumplen dependencias funcionales no triviales. En consecuencia, se puede pensar que ese tipo de combinación es una buena idea. Sin embargo, se trata de una mala idea, como puede verse si se considera el ejemplo de un empleado con dos subordinados y dos números de teléfono. Por ejemplo, sea el empleado con *id_empleado* 999999999 que tiene dos subordinados llamados "David" y "Guillermo" y dos números de teléfono, 512555123 y 512555432. En el esquema combinado hay que repetir los números de teléfono una vez por cada subordinado:

$$\begin{aligned} &(999999999, \text{David}, 512555123) \\ &(999999999, \text{David}, 512555432) \\ &(999999999, \text{Guillermo}, 512555123) \\ &(999999999, \text{Guillermo}, 512555432) \end{aligned}$$

Si no se repitieran los números de teléfono y sólo se almacenaran la primera y la última tupla, se habrían guardado el nombre de los subordinados y los números de teléfono, pero las tuplas resultantes implicarían que David correspondería al 512555123 y que Guillermo correspondería al 512555432. Como sabemos, esto es incorrecto.

Debido a que las formas normales basadas en las dependencias funcionales no son suficientes para tratar con situaciones como ésta, se han definido otras dependencias y formas normales. Se estudian en los Apartados 7.6 y 7.7.

7.4 Teoría de las dependencias funcionales

Se ha visto en los ejemplos que resulta útil poder razonar de manera sistemática sobre las dependencias funcionales como parte del proceso de comprobación de los esquemas para la FNBC o la 3NF.

7.4.1 Cierre de los conjuntos de dependencias funcionales

No es suficiente considerar el conjunto dado de dependencias funcionales. También hay que considerar *todas* las dependencias funcionales que se cumplen. Se verá que, dado un conjunto F de dependencias funcionales, se puede probar que también se cumple alguna otra dependencia funcional. Se dice que F “implica lógicamente” esas dependencias funcionales.

De manera más formal, dado un esquema relacional R , una dependencia funcional f de R está **implícada lógicamente** por un conjunto de dependencias funcionales F de R si cada ejemplar de la relación $r(R)$ que satisface F satisface también f .

Supóngase que se tiene el esquema de relación $R = (A, B, C, G, H, I)$ y el conjunto de dependencias funcionales

$$\begin{aligned} A &\rightarrow B \\ A &\rightarrow C \\ CG &\rightarrow H \\ CG &\rightarrow I \\ B &\rightarrow H \end{aligned}$$

La dependencia funcional

$$A \rightarrow H$$

está implicada lógicamente. Es decir, se puede demostrar que, siempre que el conjunto dado de dependencias funcionales se cumple en una relación, también se debe cumplir $A \rightarrow H$. Supóngase que t_1 y t_2 son tuplas tales que

$$t_1[A] = t_2[A]$$

Como se tiene que $A \rightarrow B$, se deduce de la definición de dependencia funcional que

$$t_1[B] = t_2[B]$$

Entonces, como se tiene que $B \rightarrow H$, se deduce de la definición de dependencia funcional que

$$t_1[H] = t_2[H]$$

Por tanto, se ha demostrado que, siempre que t_1 y t_2 sean tuplas tales que $t_1[A] = t_2[A]$, debe ocurrir que $t_1[H] = t_2[H]$. Pero ésa es exactamente la definición de $A \rightarrow H$.

Sea F un conjunto de dependencias funcionales. El **cierre** de F , denotado por F^+ , es el conjunto de todas las dependencias funcionales implicadas lógicamente por F . Dado F , se puede calcular F^+ directamente a partir de la definición formal de dependencia funcional. Si F fuera de gran tamaño, ese proceso sería largo y difícil. Este tipo de cálculo de F^+ requiere argumentos del tipo que se acaba de utilizar para demostrar que $A \rightarrow H$ está en el cierre del conjunto de dependencias de ejemplo.

Los **axiomas**, o reglas de inferencia, proporcionan una técnica más sencilla para el razonamiento sobre las dependencias funcionales. En las reglas que se ofrecen a continuación se utilizan las letras griegas ($\alpha, \beta, \gamma, \dots$) para los conjuntos de atributos y las letras latinas mayúsculas desde el comienzo del alfabeto para los atributos. Se emplea $\alpha\beta$ para denotar $\alpha \cup \beta$.

Se pueden utilizar las tres reglas siguientes para hallar las dependencias funcionales implicadas lógicamente. Aplicando estas reglas *repetidamente* se puede hallar todo F^+ , dado F . Este conjunto de reglas se denomina **axiomas de Armstrong** en honor de la persona que las propuso por primera vez.

- **Regla de la reflexividad.** Si α es un conjunto de atributos y $\beta \subseteq \alpha$, entonces se cumple que $\alpha \rightarrow \beta$.
- **Regla de la aumentatividad.** Si se cumple que $\alpha \rightarrow \beta$ y γ es un conjunto de atributos, entonces se cumple que $\gamma\alpha \rightarrow \gamma\beta$.
- **Regla de la transitividad.** Si se cumple que $\alpha \rightarrow \beta$ y que $\beta \rightarrow \gamma$, entonces se cumple que $\alpha \rightarrow \gamma$.

Los axiomas de Armstrong son **correctos**, ya que no generan dependencias funcionales incorrectas. Son **completos**, ya que, para un conjunto de dependencias funcionales F dado, permiten generar todo

F^+ . Las notas bibliográficas proporcionan referencias de las pruebas de su corrección y de su completitud.

Aunque los axiomas de Armstrong son completos, resulta pesado utilizarlos directamente para el cálculo de F^+ . Para simplificar más las cosas se dan unas reglas adicionales. Se pueden utilizar los axiomas de Armstrong para probar que son correctas (véanse los Ejercicios prácticos 7.4 y 7.5 y el Ejercicio 7.21).

- **Regla de la unión.** Si se cumple que $\alpha \rightarrow \beta$ y que $\alpha \rightarrow \gamma$, entonces se cumple que $\alpha \rightarrow \beta\gamma$.
- **Regla de la descomposición.** Si se cumple que $\alpha \rightarrow \beta\gamma$, entonces se cumple que $\alpha \rightarrow \beta$ y que $\alpha \rightarrow \gamma$.
- **Regla de la pseudotransitividad.** Si se cumple que $\alpha \rightarrow \beta$ y que $\gamma\beta \rightarrow \delta$, entonces se cumple que $\alpha\gamma \rightarrow \delta$.

Se aplicarán ahora las reglas al ejemplo del esquema $R = (A, B, C, G, H, I)$ y del conjunto F de dependencias funcionales $\{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$. A continuación se relacionan varios miembros de F^+ :

- $A \rightarrow H$. Dado que se cumplen $A \rightarrow B$ y $B \rightarrow H$, se aplica la regla de transitividad. Obsérvese que resultaba mucho más sencillo emplear los axiomas de Armstrong para demostrar que se cumple que $A \rightarrow H$, que deducirlo directamente a partir de las definiciones, como se ha hecho anteriormente en este apartado.
- $CG \rightarrow HI$. Dado que $CG \rightarrow H$ y $CG \rightarrow I$, la regla de la unión implica que $CG \rightarrow HI$.
- $AG \rightarrow I$. Dado que $A \rightarrow C$ y $CG \rightarrow I$, la regla de pseudotransitividad implica que se cumple que $AG \rightarrow I$.

Otra manera de averiguar que se cumple que $AG \rightarrow I$ es la siguiente. Se utiliza la regla de aumentatividad en $A \rightarrow C$ para inferir que $AG \rightarrow CG$. Aplicando la regla de transitividad a esta dependencia y a $CG \rightarrow I$, se infiere que $AG \rightarrow I$.

La Figura 7.8 muestra un procedimiento que demuestra formalmente el modo de utilizar los axiomas de Armstrong para calcular F^+ . En este procedimiento puede que la dependencia funcional ya esté presente en F^+ cuando se le añade y, en ese caso, no hay ninguna modificación en F^+ . En el Apartado 7.4.2 se verá una manera alternativa de calcular F^+ .

Los términos a la derecha y a la izquierda de las dependencias funcionales son subconjuntos de R . Dado que un conjunto de tamaño n tiene 2^n subconjuntos, hay un total de $2 \times 2^n = 2^{n+1}$ dependencias funcionales posibles, donde n es el número de atributos de R . Cada iteración del bucle repetir del procedimiento, salvo la última, añade como mínimo una dependencia funcional a F^+ . Por tanto, está garantizado que el procedimiento termine.

```

 $F^+ = F$ 
repeat
    for each dependencia funcional  $f$  de  $F^+$ 
        aplicar las reglas de reflexividad y de aumentatividad a  $f$ 
        añadir las dependencias funcionales resultantes a  $F^+$ 
    for each par de dependencias funcionales  $f_1$  y  $f_2$  de  $F^+$ 
        if  $f_1$  y  $f_2$  se pueden combinar mediante la transitividad
            añadir la dependencia funcional resultante a  $F^+$ 
until  $F^+$  deje de cambiar

```

Figura 7.8 Procedimiento para calcular F^+ .

7.4.2 Cierre de los conjuntos de atributos

Se dice que un atributo B está **determinado funcionalmente** por α si $\alpha \rightarrow B$. Para comprobar si un conjunto α es superclave hay que diseñar un algoritmo para el cálculo del conjunto de atributos determinados funcionalmente por α . Una manera de hacerlo es calcular F^+ , tomar todas las dependencias funcionales con α como término de la izquierda y tomar la unión de los términos de la derecha de todas esas dependencias. Sin embargo, hacer esto puede resultar costoso, ya que F^+ puede ser de gran tamaño.

Un algoritmo eficiente para el cálculo del conjunto de atributos determinados funcionalmente por α no sólo resulta útil para comprobar si α es superclave, sino también para otras tareas, como se verá más adelante en este apartado.

Sea α un conjunto de atributos. Al conjunto de todos los atributos determinados funcionalmente por α bajo un conjunto F de dependencias funcionales se le denomina **cierre** de α bajo F ; se denota mediante α^+ . La Figura 7.9 muestra un algoritmo, escrito en pseudocódigo, para calcular α^+ . La entrada es un conjunto F de dependencias funcionales y el conjunto α de atributos. La salida se almacena en la variable *resultado*.

Para ilustrar el modo en que trabaja el algoritmo se utilizará para calcular $(AG)^+$ con las dependencias funcionales definidas en el Apartado 7.4.1. Se comienza con $resultado = AG$. La primera vez que se ejecuta el bucle **while** para comprobar cada dependencia funcional se halla que

- $A \rightarrow B$ hace que se incluya B en *resultado*. Para comprobarlo, obsérvese que $A \rightarrow B$ se halla en F y $A \subseteq resultado$ (que es AG), por lo que $resultado := resultado \cup B$.
- $A \rightarrow C$ hace que *resultado* se transforme en $ABCG$.
- $CG \rightarrow H$ hace que *resultado* se transforme en $ABCGH$.
- $CG \rightarrow I$ hace que *resultado* se transforme en $ABCGHI$.

La segunda vez que se ejecuta el bucle **while** ya no se añade ningún atributo nuevo a *resultado*, y se termina el algoritmo.

Veamos ahora el motivo de que el algoritmo de la Figura 7.9 sea correcto. El primer paso es correcto, ya que $\alpha \rightarrow \alpha$ se cumple siempre (por la regla de reflexividad). Se afirma que, para cualquier subconjunto β de *resultado*, $\alpha \rightarrow \beta$. Dado que el bucle **while** se inicia con $\alpha \rightarrow resultado$ como cierto, sólo se puede añadir γ a *resultado* si $\beta \subseteq resultado$ y $\beta \rightarrow \gamma$. Pero, entonces, $resultado \rightarrow \beta$ por la regla de reflexividad, por lo que $\alpha \rightarrow \beta$ por transitividad. Otra aplicación de la transitividad demuestra que $\alpha \rightarrow \gamma$ (empleando $\alpha \rightarrow \beta$ y $\beta \rightarrow \gamma$). La regla de la unión implica que $\alpha \rightarrow resultado \cup \gamma$, por lo que α determina funcionalmente cualquier resultado nuevo generado en el bucle **while**. Por tanto, cualquier atributo devuelto por el algoritmo se halla en α^+ .

Resulta sencillo ver que el algoritmo halla todo α^+ . Si hay un atributo de α^+ que no se halle todavía en *resultado*, debe haber una dependencia funcional $\beta \rightarrow \gamma$ para la que $\beta \subseteq resultado$ y, como mínimo, un atributo de γ no se halla en *resultado*.

En el peor de los casos es posible que este algoritmo tarde un tiempo proporcional al cuadrado del tamaño de F . Hay un algoritmo más rápido (aunque ligeramente más complejo) que se ejecuta en un tiempo proporcional al tamaño de F ; ese algoritmo se presenta como parte del Ejercicio práctico 7.8.

Existen varias aplicaciones del algoritmo de cierre de atributos:

```

resultado :=  $\alpha$ ;
while (cambios en resultado) do
    for each dependencia funcional  $\beta \rightarrow \gamma$  in  $F$  do
        begin
            if  $\beta \subseteq resultado$  then resultado := resultado  $\cup \gamma$ ;
        fin

```

Figura 7.9 Algoritmo para el cálculo de α^+ , el cierre de α bajo F .

- Para comprobar si α es superclave, se calcula α^+ y se comprueba si contiene todos los atributos de R .
- Se puede comprobar si se cumple la dependencia funcional $\alpha \rightarrow \beta$ (o, en otras palabras, si se halla en F^+), comprobando si $\beta \subseteq \alpha^+$. Es decir, se calcula α^+ empleando el cierre de los atributos y luego se comprueba si contiene a β . Esta prueba resulta especialmente útil, como se verá más adelante en este mismo capítulo.
- Ofrece una manera alternativa de calcular F^+ : para cada $\gamma \subseteq R$ se halla el cierre γ^+ y, para cada $S \subseteq \gamma^+$, se genera la dependencia funcional $\gamma \rightarrow S$.

7.4.3 Recubrimiento canónico

Supóngase que se tiene un conjunto F de dependencias funcionales de un esquema de relación. Siempre que un usuario lleve a cabo una actualización de la relación, el sistema de bases de datos debe asegurarse de que la actualización no viole ninguna dependencia funcional, es decir, que se satisfagan todas las dependencias funcionales de F en el nuevo estado de la base de datos.

El sistema debe retroceder la actualización si viola alguna dependencia funcional del conjunto F .

Se puede reducir el esfuerzo dedicado a la comprobación de las violaciones comprobando un conjunto simplificado de dependencias funcionales que tenga el mismo cierre que el conjunto dado. Cualquier base de datos que satisfaga el conjunto simplificado de dependencias funcionales satisfará también el conjunto original y viceversa, ya que los dos conjuntos tienen el mismo cierre. Sin embargo, el conjunto simplificado resulta más sencillo de comprobar. En breve se verá el modo en que se puede crear ese conjunto simplificado. Antes, hacen falta algunas definiciones.

Se dice que un atributo de una dependencia funcional es **raro** si se puede eliminar sin modificar el cierre del conjunto de dependencias funcionales. La definición formal de los **atributos raros** es la siguiente. Considérese un conjunto F de dependencias funcionales y la dependencia funcional $\alpha \rightarrow \beta$ de F .

- El atributo A es raro en α si $A \in \alpha$ y F implica lógicamente a $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
- El atributo A es raro en β si $A \in \beta$ y el conjunto de dependencias funcionales $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ implica lógicamente a F .

Por ejemplo, supóngase que se tienen las dependencias $AB \rightarrow C$ y $A \rightarrow C$ de F . Entonces, B es raro en $AB \rightarrow C$. Como ejemplo adicional, supóngase que se tienen las dependencias funcionales $AB \rightarrow CD$ y $A \rightarrow C$ de F . Entonces, C será raro en el lado derecho de $AB \rightarrow CD$.

Hay que tener cuidado con la dirección de las implicaciones al utilizar la definición de los atributos raros: si se intercambian el lado derecho y el izquierdo, la implicación se cumplirá *siempre*. Es decir, $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$ siempre implica lógicamente a F , y F también implica lógicamente siempre a $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$.

A continuación se muestra el modo de comprobar de manera eficiente si un atributo es raro. Sea R el esquema de la relación y F el conjunto dado de dependencias funcionales que se cumplen en R . Considérese el atributo A de la dependencia $\alpha \rightarrow \beta$.

- Si $A \in \beta$, para comprobar si A es raro hay que considerar el conjunto $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ y comprobar si $\alpha \rightarrow A$ puede inferirse a partir de F' . Para ello hay que calcular α^+ (el cierre de α) bajo F' ; si α^+ incluye a A , entonces A es raro en β .
- Si $A \in \alpha$, para comprobar si A es raro, sea $\gamma = \alpha - \{A\}$, hay que comprobar si se puede inferir que $\gamma \rightarrow \beta$ a partir de F . Para ello hay que calcular γ^+ (el cierre de γ) bajo F ; si γ^+ incluye todos los atributos de β , entonces A es raro en α .

Por ejemplo, supóngase que F contiene $AB \rightarrow CD$, $A \rightarrow E$ y $E \rightarrow C$. Para comprobar si C es raro en $AB \rightarrow CD$, hay que calcular el cierre de los atributos de AB bajo $F' = \{AB \rightarrow D, A \rightarrow E\}$ y $E \rightarrow C$. El cierre es $ABCDE$, que incluye a CD , por lo que se infiere que C es raro.

$F_c = F$

repeat

 Utilizar la regla de unión para sustituir las dependencias de F_c de la forma

$\alpha_1 \rightarrow \beta_1$ y $\alpha_1 \rightarrow \beta_2$ con $\alpha_1 \rightarrow \beta_1 \beta_2$.

 Hallar una dependencia funcional $\alpha \rightarrow \beta$ de F_c con un atributo raro en α o en β .

 /* Nota: la comprobación de los atributos raros se lleva a cabo empleando F_c , no F^* */

 Si se halla algún atributo raro, hay que eliminarlo de $\alpha \rightarrow \beta$.

until F_c ya no cambie.

Figura 7.10 Cálculo del recubrimiento canónico.

El **recubrimiento canónico** F_c de F es un conjunto de dependencias tal que F implica lógicamente todas las dependencias de F_c y F_c implica lógicamente todas las dependencias de F . Además, F_c debe tener las propiedades siguientes:

- Ninguna dependencia funcional de F_c contiene atributos raros.
- El lado izquierdo de cada dependencia funcional de F_c es único. Es decir, no hay dos dependencias $\alpha_1 \rightarrow \beta_1$ y $\alpha_2 \rightarrow \beta_2$ de F_c tales que $\alpha_1 = \alpha_2$.

El recubrimiento canónico del conjunto de dependencias funcionales F puede calcularse como se muestra en la Figura 7.10. Es importante destacar que, cuando se comprueba si un atributo es raro, la comprobación utiliza las dependencias del valor actual de F_c , y **no** las dependencias de F . Si una dependencia funcional sólo contiene un atributo en su lado derecho, por ejemplo, $A \rightarrow C$, y se descubre que ese atributo es raro, se obtiene una dependencia funcional con el lado derecho vacío. Hay que eliminar esas dependencias funcionales.

Se puede demostrar que el recubrimiento canónico de F , F_c , tiene el mismo cierre que F ; por tanto, comprobar si se satisface F_c es equivalente a comprobar si se satisface F . Sin embargo, F_c es mínimo en un cierto sentido—no contiene atributos raros, y combina las dependencias funcionales con el mismo lado izquierdo. Resulta más económico comprobar F_c que comprobar el propio F .

Considérese el siguiente conjunto F de dependencias funcionales para el esquema (A, B, C) :

$$\begin{aligned} A &\rightarrow BC \\ B &\rightarrow C \\ A &\rightarrow B \\ AB &\rightarrow C \end{aligned}$$

Calcúlese el recubrimiento canónico de F .

- Hay dos dependencias funcionales con el mismo conjunto de atributos a la izquierda de la flecha:

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow B \end{aligned}$$

Estas dependencias funcionales se combinan en $A \rightarrow BC$.

- A es raro en $AB \rightarrow C$, ya que F implica lógicamente a $(F - \{AB \rightarrow C\}) \cup \{B \rightarrow C\}$. Esta aseveración es cierta porque $B \rightarrow C$ ya se halla en el conjunto de dependencias funcionales.
- C es raro en $A \rightarrow BC$, ya que $A \rightarrow BC$ está implicada lógicamente por $A \rightarrow B$ y $B \rightarrow C$.

Por tanto, el recubrimiento canónico es

$$\begin{aligned} A &\rightarrow B \\ B &\rightarrow C \end{aligned}$$

Dado un conjunto F de dependencias funcionales, puede suceder que toda una dependencia funcional del conjunto sea rara, en el sentido de que eliminarla no modifique el cierre de F . Se puede demostrar que el recubrimiento canónico F_c de F no contiene esa dependencia funcional rara. Supóngase que, por el contrario, esa dependencia rara estuviera en F_c . Los atributos del lado derecho de la dependencia serían raros, lo que no es posible por la definición de recubrimiento canónico.

Puede que el recubrimiento canónico no sea único. Por ejemplo, considérese el conjunto de dependencias funcionales $F = \{A \rightarrow BC, B \rightarrow AC \text{ y } C \rightarrow AB\}$. Si se aplica la prueba de rareza a $A \rightarrow BC$ se descubre que tanto B como C son raros bajo F . Sin embargo, sería incorrecto eliminar los dos. El algoritmo para hallar el recubrimiento canónico selecciona uno de los dos y lo elimina. Entonces:

- Si se elimina C , se obtiene el conjunto $F' = \{A \rightarrow B, B \rightarrow AC \text{ y } C \rightarrow AB\}$. Ahora B ya no es raro en el lado derecho de $A \rightarrow B$ bajo F' . Siguiendo con el algoritmo se descubre que A y B son raros en el lado derecho de $C \rightarrow AB$, lo que genera dos recubrimientos canónicos:

$$\begin{aligned} F_c &= \{A \rightarrow B, B \rightarrow C, C \rightarrow A\} \\ F_c &= \{A \rightarrow B, B \rightarrow AC, C \rightarrow B\} \end{aligned}$$

- Si se elimina B , se obtiene el conjunto $\{A \rightarrow C, B \rightarrow AC \text{ y } C \rightarrow AB\}$. Este caso es simétrico del anterior y genera dos recubrimientos canónicos:

$$\begin{aligned} F_c &= \{A \rightarrow C, C \rightarrow B, B \rightarrow A\} \\ F_c &= \{A \rightarrow C, B \rightarrow C, C \rightarrow AB\} \end{aligned}$$

Como ejercicio, el lector debe intentar hallar otro recubrimiento canónico de F .

7.4.4 Descomposición sin pérdida

Sean R un esquema de relación y F un conjunto de dependencias funcionales de R . Supóngase que R_1 y R_2 forman una descomposición de R . Sea $r(R)$ una relación con el esquema R . Se dice que la descomposición es una **descomposición sin pérdidas** si, para todos los ejemplares legales de la base de datos (es decir, los ejemplares de la base de datos que satisfacen las dependencias funcionales especificadas y otras restricciones),

$$\Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) = r$$

En otros términos, si se proyecta r sobre R_1 y R_2 y se calcula la reunión natural del resultado de la proyección, se vuelve a obtener exactamente r . Las descomposiciones que no son sin pérdidas se denominan **descomposiciones con pérdidas**. Los términos **descomposición de reunión sin pérdidas** y **descomposición de reunión con pérdidas** se utilizan a veces en lugar de descomposición sin pérdidas y descomposición con pérdidas.

Se pueden utilizar las dependencias funcionales para probar si ciertas descomposiciones son sin pérdidas. Sean R, R_1, R_2 y F como se acaban de definir. R_1 y R_2 forman una descomposición sin pérdidas de R si, como mínimo, una de las dependencias funcionales siguientes se halla en F^+ :

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

En otros términos, si $R_1 \cap R_2$ forma una superclave de R_1 o de R_2 , la descomposición de R es una descomposición sin pérdidas. Se puede utilizar el cierre de los atributos para buscar superclaves de manera eficiente, como ya se ha visto antes.

Para ilustrar esto, considérese el esquema

$$\text{prestatario_préstamo} = (\underline{\text{id_cliente}}, \underline{\text{número_préstamo}}, \text{importe})$$

que se descompuso en el Apartado 7.1.2 en

$$\begin{aligned} \text{prestatario} &= (\underline{\text{id_cliente}}, \underline{\text{número_préstamo}}) \\ \text{préstamo} &= (\underline{\text{número_préstamo}}, \text{importe}) \end{aligned}$$

```

calcular  $F^+$ ;
for each esquema  $R_i$  in  $D$  do
  begin
     $F_i :=$  la restricción de  $F^+$  a  $R_i$ ;
  end
   $F' := \emptyset$ 
  for each restricción  $F_i$  do
    begin
       $F' = F' \cup F_i$ 
    end
    calcular  $F'^+$ ;
    if ( $F'^+ = F^+$ ) then return (cierto)
    else return (falso);

```

Figura 7.11 Comprobación de la conservación de las dependencias.

En este caso $prestatario \cap préstamo = número_préstamo$ y $número_préstamo \rightarrow importe$, lo que satisface la regla de la descomposición sin pérdidas.

Para el caso general de la descomposición simultánea de un esquema en varios, la comprobación de la descomposición sin pérdidas es más complicada. Véanse las notas bibliográficas para tener referencias sobre este tema.

Mientras que la comprobación de la descomposición binaria es, claramente, una condición suficiente para la descomposición sin pérdidas, sólo se trata de una condición necesaria si todas las restricciones son dependencias funcionales. Más adelante se verán otros tipos de restricciones (especialmente, un tipo de restricción denominada dependencia multivalorada, que se estudia en el Apartado 7.6.1), que pueden garantizar que una descomposición sea sin pérdidas aunque no esté presente ninguna dependencia funcional.

7.4.5 Conservación de las dependencias

Resulta más sencillo caracterizar la conservación de las dependencias empleando la teoría de las dependencias funcionales que mediante el enfoque ad hoc que se utilizó en el Apartado 7.3.3.

Sea F un conjunto de dependencias funcionales del esquema R y R_1, R_2, \dots, R_n una descomposición de R . La **restricción** de F a R_i es el conjunto F_i de todas las dependencias funcionales de F^+ que sólo incluyen atributos de R_i . Dado que todas las dependencias funcionales de cada restricción implican a los atributos de un único esquema de relación, es posible comprobar el cumplimiento de esas dependencias verificando sólo una relación.

Obsérvese que la definición de restricción utiliza todas las dependencias de F^+ , no sólo las de F . Por ejemplo, supóngase que se tiene $F = \{A \rightarrow B, B \rightarrow C\}$ y que se tiene una descomposición en AC y AB . La restricción de F a AC es, por tanto, $A \rightarrow C$, ya que $A \rightarrow C$ se halla en F^+ , aunque no se halle en F .

El conjunto de restricciones F_1, F_2, \dots, F_n es el conjunto de dependencias que pueden comprobarse de manera eficiente. Ahora cabe preguntarse si es suficiente comprobar sólo las restricciones. Sea $F' = F_1 \cup F_2 \cup \dots \cup F_n$. F' es un conjunto de dependencias funcionales del esquema R pero, en general $F' \neq F$. Sin embargo, aunque $F' \neq F$, puede ocurrir que $F'^+ = F^+$. Si esto último es cierto, entonces, todas las dependencias de F están implicadas lógicamente por F' y, si se comprueba que se satisface F' , se habrá comprobado que se satisface F . Se dice que las descomposiciones que tienen la propiedad $F'^+ = F^+$ son **descomposiciones que conservan las dependencias**.

La Figura 7.11 muestra un algoritmo para la comprobación de la conservación de las dependencias. La entrada es el conjunto $D = \{R_1, R_2, \dots, R_n\}$ de esquemas de relaciones descompuestas y el conjunto F de dependencias funcionales. Este algoritmo resulta costoso, ya que exige el cálculo de F^+ . En lugar de aplicar el algoritmo de la Figura 7.11, se consideran dos alternativas.

En primer lugar, hay que tener en cuenta que si se puede comprobar cada miembro de F en una de las relaciones de la descomposición, la descomposición conserva las dependencias. Se trata de una manera sencilla de demostrar la conservación de las dependencias; no obstante, no funciona siempre. Hay casos en los que, aunque la descomposición conserve las dependencias, hay alguna dependencia de F que no

se puede comprobar para ninguna relación de la descomposición. Por tanto, esta prueba alternativa sólo se puede utilizar como condición suficiente que es fácil de comprobar; si falla, no se puede concluir que la descomposición no conserve las dependencias; en vez de eso, hay que aplicar la prueba general.

A continuación se da una comprobación alternativa de la conservación de las dependencias, que evita tener que calcular F^+ . La idea intuitiva subyacente a esta comprobación se explicará tras presentarla. La comprobación aplica el procedimiento siguiente a cada $\alpha \rightarrow \beta$ de F .

```

resultado =  $\alpha$ 
while (cambios en resultado) do
    for each  $R_i$  de la descomposición
         $t = (\text{resultado} \cap R_i)^+ \cap R_i$ 
        resultado = resultado  $\cup t$ 

```

En este caso, el cierre de los atributos se halla bajo el conjunto de dependencias funcionales F . Si *resultado* contiene todos los atributos de β , se conserva la dependencia funcional $\alpha \rightarrow \beta$. La descomposición conserva las dependencias si, y sólo si, el procedimiento prueba que se conservan todas las dependencias de F .

Las dos ideas claves subyacentes a la comprobación anterior son las siguientes.

- La primera idea es comprobar cada dependencia funcional $\alpha \rightarrow \beta$ de F para ver si se conserva en F' (donde F' es tal y como se define en la Figura 7.11). Para ello, se calcula el cierre de α bajo F' ; la dependencia se conserva exactamente cuando el cierre incluye a β . La descomposición conserva la dependencia si (y sólo si) se comprueba que se conservan todas las dependencias de F .
- La segunda idea es emplear una forma modificada del algoritmo de cierre de atributos para calcular el cierre bajo F' , sin llegar a calcular antes F' . Se desea evitar el cálculo de F' , ya que resulta bastante costoso. Téngase en cuenta que F' es la unión de las F_i , donde F_i es la restricción de F sobre R_i . El algoritmo calcula el cierre de los atributos de $(\text{resultado} \cap R_i)$ con respecto a F , intersecta el cierre con R_i y añade el conjunto de atributos resultante a *resultado*; esta secuencia de pasos es equivalente al cálculo del cierre de *resultado* bajo F_i . La repetición de este paso para cada i del interior del bucle genera el cierre de *resultado* bajo F' .

Para comprender el motivo de que este enfoque modificado al cierre de atributos funcione correctamente, hay que tener en cuenta que para cualquier $\gamma \subseteq R_i$, $\gamma \rightarrow \gamma^+$ es una dependencia funcional de F^+ , y que $\gamma \rightarrow \gamma^+ \cap R_i$ es una dependencia funcional que se halla en F_i , la restricción de F^+ a R_i . A la inversa, si $\gamma \rightarrow \delta$ estuvieran en F_i , δ sería un subconjunto de $\gamma^+ \cap R_i$.

Esta comprobación tarda un tiempo que es polinómico, en lugar del exponencial necesario para calcular F^+ .

7.5 Algoritmos de descomposición

Los esquemas de bases de datos del mundo real son mucho mayores que los ejemplos que caben en las páginas de un libro. Por este motivo, hacen falta algoritmos para la generación de diseños que se hallen en la forma normal adecuada. En este apartado se presentan algoritmos para la FNBC y para la 3NF.

7.5.1 Descomposición en la FNBC

Se puede emplear la definición de la FNBC para comprobar directamente si una relación se halla en esa forma normal. Sin embargo, el cálculo de F^+ puede resultar una tarea tediosa. En primer lugar se van a describir pruebas simplificadas para verificar si una relación dada se halla en la FNBC. En caso de que no lo esté, se puede descomponer para crear relaciones que sí estén en la FNBC. Más avanzado este apartado se describirá un algoritmo para crear descomposiciones sin pérdidas de las relaciones, de modo que esas descomposiciones se hallen en la FNBC.

7.5.1.1 Comprobación de la FNBC

La comprobación de relaciones para ver si satisfacen la FNBC se puede simplificar en algunos casos:

- Para comprobar si la dependencia no trivial $\alpha \rightarrow \beta$ provoca alguna violación de la FNBC hay que calcular α^+ (el cierre de los atributos de α) y comprobar si incluye todos los atributos de R ; es decir, si es superclave de R .
- Para comprobar si el esquema de relación R se halla en la FNBC basta con comprobar sólo si las dependencias del conjunto F dado violan la FNBC, en vez de comprobar todas las dependencias de F^+ .

Se puede probar que, si ninguna de las dependencias de F provoca violaciones de la FNBC, ninguna de las dependencias de F^+ lo hace tampoco.

Por desgracia, el último procedimiento no funciona cuando se descomponen relaciones. Es decir, en las descomposiciones *no* basta con emplear F cuando se comprueba si la relación R_i de R viola la FNBC. Por ejemplo, considérese el esquema de relación $R(A, B, C, D, E)$, con las dependencias funcionales F que contienen $A \rightarrow B$ and $BC \rightarrow D$. Supóngase que se descompusiera en $R_1(A, B)$ y en $R_2(A, C, D, E)$. Ahora ninguna de las dependencias de F contiene únicamente atributos de (A, C, D, E) , por lo que se podría creer erróneamente que R_2 satisface la FNBC. De hecho, hay una dependencia $AC \rightarrow D$ de F^+ (que se puede inferir empleando la regla de la pseudotransitividad con las dos dependencias de F), que prueba que R_2 no se halla en la FNBC. Por tanto, puede que haga falta una dependencia que esté en F^+ , pero no en F , para probar que una relación descompuesta no se halla en la FNBC.

Una comprobación alternativa de la FNBC resulta a veces más sencilla que calcular todas las dependencias de F^+ . Para comprobar si una relación R_i de una descomposición de R se halla en la FNBC, se aplica esta comprobación:

- Para cada subconjunto α de atributos de R_i se comprueba que α^+ (el cierre de los atributos de α bajo F) no incluye ningún atributo de $R_i - \alpha$ o bien incluye todos los atributos de R_i .

Si algún conjunto de atributos α de R_i viola esta condición, considérese la siguiente dependencia funcional, que se puede probar que se halla en F^+ :

$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i.$$

La dependencia anterior muestra que R_i viola la FNBC.

7.5.1.2 Algoritmo de descomposición de la FNBC

Ahora se puede exponer un método general para descomponer los esquemas de relación de manera que satisfagan la FNBC. La Figura 7.12 muestra un algoritmo para esta tarea. Si R no está en la FNBC, se puede descomponer en un conjunto de esquemas en la FNBC, R_1, R_2, \dots, R_n utilizando este algoritmo. El algoritmo utiliza las dependencias que demuestran la violación de la FNBC para llevar a cabo la descomposición.

```

resultado := {R};
hecho := falso;
calcular F+;
while (not hecho) do
    if (hay algún esquema  $R_i$  de resultado que no se halle en la FNBC)
        then begin
            sea  $\alpha \rightarrow \beta$  una dependencia funcional no trivial que se cumple
            en  $R_i$  tal que  $\alpha \rightarrow R_i$  no se halla en  $F^+$ , y  $\alpha \cap \beta = \emptyset$ ;
            resultado := (resultado -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
        end
    else hecho := cierto;

```

Figura 7.12 Algoritmo de descomposición en la FNBC.

La descomposición que genera este algoritmo no sólo está en la FNBC, sino que también es una descomposición sin pérdidas. Para ver el motivo de que el algoritmo sólo genere descomposiciones sin pérdidas hay que darse cuenta de que, cuando se sustituye el esquema R_i por $((R_i - \beta) \cup (\alpha, \beta))$, se cumple que $\alpha \rightarrow \beta$ y que $(R_i - \beta) \cap (\alpha, \beta) = \alpha$.

Si no se exigiera que $\alpha \cap \beta = \emptyset$, los atributos de $\alpha \cap \beta$ no aparecerían en el esquema $((R_i - \beta) \cup (\alpha, \beta))$, y ya no se cumpliría la dependencia $\alpha \rightarrow \beta$.

Es fácil ver que la descomposición de *prestatario_préstamo* del Apartado 7.3.2 se obtiene de la aplicación del algoritmo. La dependencia funcional *número_préstamo* \rightarrow *importe* satisface la condición $\alpha \cap \beta = \emptyset$ y, por tanto, se elige para descomponer el esquema.

El algoritmo de descomposición en la FNBC tarda un tiempo exponencial en relación con el tamaño del esquema inicial, ya que el algoritmo para comprobar si las relaciones de la descomposición satisfacen la FNBC puede tardar un tiempo exponencial. Las notas bibliográficas ofrecen referencias de un algoritmo que puede calcular la descomposición en la FNBC en un tiempo polinómico. Sin embargo, puede que ese algoritmo “sobrenormalice”, es decir, descomponga relaciones sin que sea necesario.

A modo de ejemplo, más prolífico, del empleo del algoritmo de descomposición en la FNBC, supóngase que se tiene un diseño de base de datos que emplea el siguiente esquema *empréstito*:

$$\text{empréstito} = (\text{nombre_sucursal}, \text{ciudad_sucursal}, \text{activos}, \text{nombre_cliente}, \text{número_préstamo}, \text{importe})$$

El conjunto de dependencias funcionales que se exige que se cumplan en *empréstito* es

$$\begin{aligned} \text{nombre_sucursal} &\rightarrow \text{activos ciudad_sucursal} \\ \text{número_préstamo} &\rightarrow \text{importe nombre_sucursal} \end{aligned}$$

Una clave candidata para este esquema es $\{\text{número_préstamo}, \text{nombre_cliente}\}$.

Se puede aplicar el algoritmo de la Figura 7.12 al ejemplo *empréstito* de la manera siguiente:

- La dependencia funcional

$$\text{nombre_sucursal} \rightarrow \text{activos ciudad_sucursal}$$

se cumple, pero *nombre_sucursal* no es superclave. Por tanto, *empréstito* no se halla en la FNBC. Se sustituye *empréstito* por

$$\begin{aligned} \text{sucursal} &= (\text{nombre_sucursal}, \text{ciudad_sucursal}, \text{activos}) \\ \text{info_préstamo} &= (\text{nombre_sucursal}, \text{nombre_cliente}, \text{número_préstamo}, \text{importe}) \end{aligned}$$

- Entre las pocas dependencias funcionales no triviales que se cumplen en *sucursal* está *nombre_sucursal*, al lado izquierdo de la flecha. Dado que *nombre_sucursal* es clave de *sucursal*, la relación *sucursal* se halla en la FNBC.
- La dependencia funcional

$$\text{número_préstamo} \rightarrow \text{importe nombre_sucursal}$$

se cumple en *info_préstamo*, pero *número_préstamo* no es clave de *info_préstamo*. Se sustituye *info_préstamo* por

$$\begin{aligned} \text{sucursal_préstamo} &= (\text{número_préstamo}, \text{nombre_sucursal}, \text{importe}) \\ \text{prestatario} &= (\text{nombre_cliente}, \text{número_préstamo}) \end{aligned}$$

- *sucursal_préstamo* y *prestatario* están en la FNBC.

Por tanto, la descomposición de *empréstito* da lugar a los tres esquemas de relación *sucursal*, *sucursal_préstamo* y *prestatario*, cada uno de los cuales se halla en la FNBC. Se puede comprobar que la descomposición es sin pérdidas y que conserva las dependencias.

```

sea  $F_c$  un recubrimiento canónico de  $F$ ;
i := 0;
for each dependencia funcional  $\alpha \rightarrow \beta$  de  $F_c$  do
  if ninguno de los esquemas  $R_j, j = 1, 2, \dots, i$  contiene  $\alpha \beta$ 
    then begin
       $i := i + 1$ ;
       $R_i := \alpha \beta$ ;
    end
  if ninguno de los esquemas  $R_j, j = 1, 2, \dots, i$  contiene una clave candidata de  $R$ 
    then begin
       $i := i + 1$ ;
       $R_i :=$  cualquier clave candidata de  $R$ ;
    end
  return  $(R_1, R_2, \dots, R_i)$ 

```

Figura 7.13 Descomposición en la 3NF sin pérdidas que conserva las dependencias.

Téngase en cuenta que, aunque el esquema *sucursal_préstamo* anterior se halle en la FNBC, se puede decidir descomponerlo aún más empleando la dependencia funcional *número_préstamo* \rightarrow *importe* para obtener los esquemas

$$\text{préstamo} = (\text{número_préstamo}, \text{importe}) \quad \text{sucursal_préstamo} = (\text{número_préstamo}, \text{nombre_sucursal})$$

Estos esquemas corresponden a los que se han utilizado en este capítulo.

7.5.2 Descomposición en la 3FN

La Figura 7.13 muestra un algoritmo para la búsqueda de descomposiciones en la 3FN sin pérdidas y que conserven las dependencias. El conjunto de dependencias F_c utilizado en el algoritmo es un recubrimiento canónico de F . Obsérvese que el algoritmo considera el conjunto de esquemas $R_j, j = 1, 2, \dots, i$; inicialmente $i = 0$ y, en ese caso, el conjunto está vacío.

Se aplicará este algoritmo al ejemplo del Apartado 7.3.3 en el que se demostró que

$$\text{sucursal_asesor} = (\text{id_cliente}, \text{id_empleado}, \text{nombre_sucursal}, \text{tipo})$$

se halla en la 3NF, aunque no se halle en la FNBC. El algoritmo utiliza las dependencias funcionales de F , que, en este caso, también son F_c :

$$\begin{aligned} &(\text{id_cliente}, \text{id_empleado} \rightarrow \text{nombre_sucursal}, \text{tipo}) \\ &\text{id_empleado} \rightarrow \text{nombre_sucursal} \end{aligned}$$

y considera dos esquemas en el bucle **for**. A partir de la primera dependencia funcional el algoritmo genera como R_1 el esquema $(\text{id_cliente}, \text{id_empleado}, \text{nombre_sucursal}, \text{tipo})$. A partir de la segunda, el algoritmo genera el esquema $(\text{id_empleado}, \text{nombre_sucursal})$, pero no lo crea como R_2 , ya que se halla incluido en R_1 y, por tanto, falla la condición **if**.

El algoritmo garantiza la conservación de las dependencias mediante la creación explícita de un esquema para cada dependencia del recubrimiento canónico. Asegura que la descomposición sea sin pérdidas al garantizar que, como mínimo, un esquema contenga una clave candidata del esquema que se está descomponiendo. El Ejercicio práctico 7.12 proporciona algunos indicios de la prueba de que esto basta para garantizar una descomposición sin pérdidas.

Este algoritmo también se denomina **algoritmo de síntesis de la 3NF**, ya que toma un conjunto de dependencias y añade los esquemas de uno en uno, en lugar de descomponer el esquema inicial de manera repetida. El resultado no queda definido de manera única, ya que cada conjunto de dependencias funcionales puede tener más de un recubrimiento canónico y, además, en algunos casos, el resultado del algoritmo depende del orden en que considere las dependencias de F_c .

Si una relación R_i está en la descomposición generada por el algoritmo de síntesis, entonces R_i está en la 3FN. Recuérdese que, cuando se busca la 3FN, basta con considerar las dependencias funcionales cuyo lado derecho sea un solo atributo. Por tanto, para ver si R_i está en la 3FN, hay que convencerse de que cualquier dependencia funcional $\gamma \rightarrow B$ que se cumpla en R_i satisface la definición de la 3FN.

Supóngase que la dependencia que generó R_i en el algoritmo de síntesis es $\alpha \rightarrow \beta$. Ahora bien, B debe estar en α o en β , ya que B está en R_i y $\alpha \rightarrow \beta$ ha generado R_i . Considérense los tres casos posibles:

- B está tanto en α como en β . En ese caso, la dependencia $\alpha \rightarrow \beta$ no habría estado en F_c , ya que B sería rara en β . Por tanto, este caso no puede darse.
- B está en β pero no en α . Considérense dos casos:
 - γ es superclave. Se satisface la segunda condición de la 3FN.
 - γ no es superclave. Entonces, α debe contener algún atributo que no se halle en γ . Ahora bien, como $\gamma \rightarrow B$ se halla en F^+ , debe poder obtenerse a partir de F_c mediante el algoritmo del cierre de atributos de γ . La obtención no puede haber empleado $\alpha \rightarrow \beta$ —si lo hubiera hecho, α debería estar contenida en el cierre de los atributos de γ , lo que no es posible, ya que se ha dado por supuesto que γ no es superclave. Ahora bien, empleando $\alpha \rightarrow (\beta - \{B\})$ y $\gamma \rightarrow B$, se puede obtener que $\alpha \rightarrow B$ (debido a que $\gamma \subseteq \alpha\beta$ y a que γ no puede contener a B porque $\gamma \rightarrow B$ no es trivial). Esto implicaría que B es raro en el lado derecho de $\alpha \rightarrow \beta$, lo que no es posible, ya que $\alpha \rightarrow \beta$ está en el recubrimiento canónico F_c . Por tanto, si B está en β , γ debe ser superclave, y se debe satisfacer la segunda condición de la 3FN.
- B está en α pero no en β .

Como α es clave candidata, se satisface la tercera alternativa de la definición de la 3FN.

Es interesante que el algoritmo que se ha descrito para la descomposición en la 3FN pueda implementarse en tiempo polinómico, aunque la comprobación de una relación dada para ver si satisface la 3FN sea NP-dura (lo que significa que es muy improbable que se invente nunca un algoritmo de tiempo polinómico para esta tarea).

7.5.3 Comparación de la FNBC y la 3FN

De las dos formas normales para los esquemas de las bases de datos relacionales, la 3FN y la FNBC, la 3FN es más conveniente, ya que se sabe que siempre es posible obtener un diseño en la 3FN sin sacrificar la ausencia de pérdidas ni la conservación de las dependencias. Sin embargo, la 3FN presenta inconvenientes: puede que haya que emplear valores nulos para representar algunas de las relaciones significativas posibles entre los datos y existe el problema de la repetición de la información.

Los objetivos del diseño de bases de datos con dependencias funcionales son:

1. FNBC
2. Ausencia de pérdidas
3. Conservación de las dependencias

Como no siempre resulta posible satisfacer las tres, puede que nos veamos obligados a escoger entre la FNBC y la conservación de las dependencias con la 3FN.

Merece la pena destacar que el SQL no ofrece una manera de especificar las dependencias funcionales, salvo para el caso especial de la declaración de las superclaves mediante las restricciones **primary key** o **unique**. Es posible, aunque un poco complicado, escribir asertos que hagan que se cumpla una dependencia funcional determinada (véase el Ejercicio práctico 7.9); por desgracia, la comprobación de los asertos resultaría muy costosa en la mayor parte de los sistemas de bases de datos. Por tanto, aunque se tenga una descomposición que conserve las dependencias, si se utiliza el SQL estándar, sólo se podrán comprobar de manera eficiente las dependencias funcionales cuyo lado izquierdo sea una clave.

Aunque puede que la comprobación de las dependencias funcionales implique una reunión si la descomposición no conserva las dependencias, se puede reducir su coste empleando vistas materializadas, que se pueden utilizar en la mayor parte de los sistemas de bases de datos. Dada una descomposición en la FNBC que no conserve las dependencias, se considera cada dependencia de un recubrimiento mínimo F_c que no se conserva en la descomposición. Para cada una de esas dependencias $\alpha \rightarrow \beta$, se define una vista materializada que calcula una reunión de todas las relaciones de la descomposición y proyecta el resultado sobre $\alpha\beta$. La dependencia funcional puede comprobarse fácilmente en la vista materializada

mediante una restricción **unique**(α). La parte negativa es que hay una sobrecarga espacial y temporal debida a la vista materializada, pero la positiva es que el programador de la aplicación no tiene que preocuparse de escribir código para hacer que los datos redundantes se conserven consistentes en las actualizaciones; es labor del sistema de bases de datos conservar la vista materializada, es decir, mantenerla actualizada cuando se actualice la base de datos (más adelante, en el Apartado 14.5, se describe el modo en que el sistema de bases de datos puede llevar a cabo de manera eficiente el mantenimiento de las vistas materializadas).

Por tanto, en caso de que no se pueda obtener una descomposición en la FNBC que conserve las dependencias, suele resultar preferible optar por la 3NF y emplear técnicas como las vistas materializadas para reducir el coste de la comprobación de las dependencias funcionales.

7.6 Descomposición mediante dependencias multivaloradas

No parece que algunos esquemas de relación, aunque se hallen en la FNBC, estén suficientemente normalizados, en el sentido de que siguen sufriendo el problema de la repetición de información. Considérese nuevamente el ejemplo bancario. Supóngase que, en un diseño alternativo del esquema de la base de datos bancaria, se tiene el esquema

$$\text{préstamo_cliente} = (\underline{\text{número_préstamo}}, \underline{\text{id_cliente}}, \text{nombre_cliente}, \text{calle_cliente}, \text{ciudad_cliente})$$

El lector avisado reconocerá este esquema como no correspondiente a la FNBC, debido a la dependencia funcional

$$\text{id_cliente} \rightarrow \text{nombre_cliente}, \text{calle_cliente}, \text{ciudad_cliente}$$

y a que id_cliente no es clave de préstamo_cliente . No obstante, supóngase que el banco está atrayendo clientes ricos que tienen varios domicilios (por ejemplo, una residencia de invierno y otra de verano). Entonces, ya no se desea que se cumpla la dependencia funcional $\text{id_cliente} \rightarrow \text{calle_cliente}, \text{ciudad_cliente}$, aunque, por supuesto, se sigue deseando que se cumpla $\text{id_cliente} \rightarrow \text{nombre_cliente}$ (es decir, el banco no trata con clientes que operan con varios alias). A partir del algoritmo de descomposición en la FNBC se obtienen dos esquemas:

$$R_1 = (\text{id_cliente}, \text{nombre_cliente})$$

$$R_2 = (\underline{\text{número_préstamo}}, \underline{\text{id_cliente}}, \text{calle_cliente}, \text{ciudad_cliente})$$

Los dos se encuentran en la FNBC (recuérdese que no sólo puede cada cliente tener concedido más de un préstamo, sino que también se pueden conceder préstamos a grupos de personas y, por tanto, no se cumplen ni $\text{id_cliente} \rightarrow \text{número_préstamo}$ ni $\text{número_préstamo} \rightarrow \text{id_cliente}$).

Pese a que R_2 se halla en la FNBC, hay redundancia. Se repite la dirección de cada residencia de cada cliente una vez por cada préstamo que tenga concedido ese cliente. Este problema se puede resolver descomponiendo más aún R_2 en:

$$\text{id_cliente_préstamo} = (\text{número_préstamo}, \text{id_cliente})$$

$$\text{residencia_cliente} = (\text{id_cliente}, \text{calle_cliente}, \text{ciudad_cliente})$$

pero no hay ninguna restricción que nos lleve a hacerlo.

Para tratar este problema hay que definir una nueva modalidad de restricción, denominada **dependencia multivalorada**. Al igual que se hizo con las dependencias funcionales, se utilizarán las dependencias multivaloradas para definir una forma normal para los esquemas de relación. Esta forma normal, denominada **cuarta forma normal** (4FN), es más restrictiva que la FNBC. Se verá que cada esquema en la 4FN también se halla en la FNBC, pero que hay esquemas en la FNBC que no se hallan en la 4FN.

7.6.1 Dependencias multivaloradas

Las dependencias funcionales impiden que ciertas tuplas estén en una relación dada. Si $A \rightarrow B$, entonces no puede haber dos tuplas con el mismo valor de A y diferentes valores de B . Las dependencias

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Figura 7.14 Representación tabular de $\alpha \rightarrow\!\!\!\rightarrow \beta$.

multivaloradas, por otro lado, no impiden la existencia de esas tuplas. Por el contrario, exigen que estén presentes en la relación otras tuplas de una forma determinada. Por este motivo, las dependencias funcionales se denominan a veces **dependencias que generan igualdades**; y las dependencias multivaloradas, **dependencias!que generan tuplas**.

Sea R un esquema de relación y sean $\alpha \subseteq R$ y $\beta \subseteq R$. La **dependencia multivalorada**

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

se cumple en R si, en cualquier relación legal $r(R)$ para todo par de tuplas t_1 y t_2 de r tales que $t_1[\alpha] = t_2[\alpha]$ existen unas tuplas t_3 y t_4 de r tales que

$$\begin{aligned}t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\t_3[\beta] &= t_1[\beta] \\t_3[R - \beta] &= t_2[R - \beta] \\t_4[\beta] &= t_2[\beta] \\t_4[R - \beta] &= t_1[R - \beta]\end{aligned}$$

Esta definición es menos complicada de lo que parece. La Figura 7.14 muestra una representación tabular de t_1 , t_2 , t_3 y t_4 . De manera intuitiva, la dependencia multivalorada $\alpha \rightarrow\!\!\!\rightarrow \beta$ indica que la relación entre α y β es independiente de la relación entre α y $R - \beta$. Si todas las relaciones del esquema R satisfacen la dependencia multivalorada $\alpha \rightarrow\!\!\!\rightarrow \beta$, entonces $\alpha \rightarrow\!\!\!\rightarrow \beta$ es una dependencia multivalorada *trivial* del esquema R . Por tanto, $\alpha \rightarrow\!\!\!\rightarrow \beta$ es trivial si $\beta \subseteq \alpha$ o $\beta \cup \alpha = R$.

Para ilustrar la diferencia entre las dependencias funcionales y las multivaloradas, considérense de nuevo el esquema R_2 y la relación de ejemplo de ese esquema mostrada en la Figura 7.15. Hay que repetir el número de préstamo una vez por cada dirección que tenga el cliente, y la dirección en cada préstamo que tenga concedido ese cliente. Esta repetición es innecesaria, ya que la relación entre cada cliente y su dirección es independiente de la relación entre ese cliente y el préstamo. Si un cliente con identificador de cliente 99123 tiene concedido un préstamo (por ejemplo, el préstamo número P-23), se desea que ese préstamo esté asociado con todas las direcciones de ese cliente. Por tanto, la relación de la Figura 7.16 es ilegal. Para hacer que esa relación sea legal hay que añadir a la relación de la Figura 7.16 las tuplas (P-23, 99123, Mayor, Chinchón) y (P-27, 99123, Carretas, Cerceda).

Si se compara el ejemplo anterior con la definición de dependencia multivalorada, se ve que se desea que se cumpla la dependencia multivalorada

$$id_cliente \rightarrow\!\!\!\rightarrow calle_cliente\ ciudad_cliente$$

También se cumplirá la dependencia multivalorada $id_cliente \rightarrow\!\!\!\rightarrow numero_prestamo$. Pronto se verá que son equivalentes.

Al igual que las dependencias funcionales, las dependencias multivaloradas se utilizan de dos maneras:

<i>numero_prestamo</i>	<i>id_cliente</i>	<i>calle_cliente</i>	<i>ciudad_cliente</i>
P-23	99123	Carretas	Cerceda
P-23	99123	Mayor	Chinchón
P-93	15106	Leganitos	Aluche

Figura 7.15 Ejemplo de redundancia en un esquema en la FNBC.

<u>número_préstamo</u>	<u>id_cliente</u>	<u>calle_cliente</u>	<u>ciudad_cliente</u>
P-23	99123	Carretas	Cerceda
P-27	99123	Mayor	Chinchón

Figura 7.16 La relación ilegal R_2 .

1. Para verificar las relaciones y determinar si son legales bajo un conjunto dado de dependencias funcionales y multivaloradas.
2. Para especificar restricciones del conjunto de relaciones legales; de este modo, sólo habrá que preocuparse de las relaciones que satisfagan un conjunto dado de dependencias funcionales y multivaloradas.

Téngase en cuenta que, si una relación r no satisface una dependencia multivalorada dada, se puede crear una relación r' que sí la satisfaga añadiendo tuplas a r .

Supóngase que D denota un conjunto de dependencias funcionales y multivaloradas. El **cierre** D^+ de D es el conjunto de todas las dependencias funcionales y multivaloradas implicadas lógicamente por D . Al igual que se hizo con las dependencias funcionales, se puede calcular D^+ a partir de D , empleando las definiciones formales de dependencia funcional y multivalorada. Con este razonamiento se puede trabajar con dependencias multivaloradas muy sencillas. Afortunadamente, parece que las dependencias multivaloradas que se dan en la práctica son bastante sencillas. Para dependencias complejas es mejor razonar con conjuntos de dependencias mediante un sistema de reglas de inferencia (el Apartado C.1.1 del apéndice describe un sistema de reglas de inferencia para las dependencias multivaloradas).

A partir de la definición de dependencia multivalorada se puede obtener la regla siguiente:

- Si $\alpha \rightarrow \beta$, entonces $\alpha \rightarrow\rightarrow \beta$.

En otras palabras, todas las dependencias funcionales son también dependencias multivaloradas.

7.6.2 Cuarta forma normal

Considérese nuevamente el ejemplo del esquema en la FNBC

$$R_2 = (\underline{\text{número_préstamo}}, \underline{\text{id_cliente}}, \underline{\text{calle_cliente}}, \underline{\text{ciudad_cliente}})$$

en el que se cumple la dependencia multivalorada $\text{id_cliente} \rightarrow\rightarrow \text{calle_cliente}$ ciudad_cliente . Se vio en los primeros párrafos del Apartado 7.6 que, aunque este esquema se halla en la FNBC, el diseño no es el ideal, ya que hay que repetir la información sobre la dirección del cliente para cada préstamo. Se verá que se puede utilizar esta dependencia multivalorada para mejorar el diseño de la base de datos, realizando la descomposición del esquema en la **cuarta forma normal**.

Un esquema de relación R está en la **cuarta forma normal** (4FN) con respecto a un conjunto D de dependencias funcionales y multivaloradas si, para todas las dependencias multivaloradas de D^+ de la forma $\alpha \rightarrow\rightarrow \beta$, donde $\alpha \subseteq R$ y $\beta \subseteq R$, se cumple, como mínimo, una de las condiciones siguientes

- $\alpha \rightarrow\rightarrow \beta$ es una dependencia multivalorada trivial.
- α es superclave del esquema R .

El diseño de una base de datos está en la 4FN si cada componente del conjunto de esquemas de relación que constituye el diseño se halla en la 4FN.

Téngase en cuenta que la definición de la 4FN sólo se diferencia de la definición de la FNBC en el empleo de las dependencias multivaloradas en lugar de las dependencias funcionales. Todos los esquemas en la 4FN están en la FNBC. Para verlo hay que darse cuenta de que, si un esquema R no se halla en la FNBC, hay una dependencia funcional no trivial $\alpha \rightarrow \beta$ que se cumple en R en la que α no es superclave. Como $\alpha \rightarrow \beta$ implica $\alpha \rightarrow\rightarrow \beta$, R no puede estar en la 4FN.

```

resultado := {R};
hecho := falso;
calcular  $D^+$ ; dado el esquema  $R_i$ ,  $D_i$  denotará la restricción de  $D^+$  a  $R_i$ 
while (not hecho) do
  if (hay un esquema  $R_i$  en resultado que no se halla en la 4FN con respecto a  $D_i$ )
    then begin
      sea  $\alpha \rightarrow\!\!\! \rightarrow \beta$  una dependencia multivalorada no trivial que se cumple
      en  $R_i$  tal que  $\alpha \rightarrow R_i$  no se halla en  $D_i$ , y  $\alpha \cap \beta = \emptyset$ ;
      resultado := (resultado -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
    end
  else hecho := cierto;

```

Figura 7.17 Algoritmo de descomposición en la 4FN.

Sea R un esquema de relación y sean R_1, R_2, \dots, R_n una descomposición de R . Para comprobar si cada esquema de relación R_i de la descomposición se halla en la 4FN hay que averiguar las dependencias multivaloradas que se cumplen en cada R_i . Recuérdese que, para un conjunto F de dependencias funcionales, la restricción F_i de F a R_i son todas las dependencias funcionales de F^+ que sólo incluyen los atributos de R_i . Considérese ahora un conjunto D de dependencias funcionales y multivaloradas. La **restricción** de D a R_i es el conjunto D_i consistente en

1. Todas las dependencias funcionales de D^+ que sólo incluyen atributos de R_i
2. Todas las dependencias multivaloradas de la forma

$$\alpha \rightarrow\!\!\! \rightarrow \beta \cap R_i$$

donde $\alpha \subseteq R_i$ y $\alpha \rightarrow\!\!\! \rightarrow \beta$ está en D^+ .

7.6.3 Descomposición en la 4FN

La analogía entre la 4FN y la FNBC es aplicable al algoritmo para descomponer esquemas en la 4FN. La Figura 7.17 muestra el algoritmo de descomposición en la 4FN. Es idéntico al algoritmo de descomposición en la FNBC de la Figura 7.12, salvo en que emplea dependencias multivaloradas en lugar de funcionales y en que utiliza la restricción de D^+ a R_i .

Si se aplica el algoritmo de la Figura 7.17 a $(número_préstamo, id_cliente, calle_cliente, ciudad_cliente)$, se descubre que $id_cliente \rightarrow\!\!\! \rightarrow número_préstamo$ es una dependencia multivalorada no trivial, y que $id_cliente$ no es superclave del esquema. De acuerdo con el algoritmo, se sustituye el esquema original por estos dos esquemas:

$$\begin{aligned} id_cliente_préstamo &= (número_préstamo, id_cliente) \\ residencia_cliente &= (id_cliente, calle_cliente, ciudad_cliente) \end{aligned}$$

Este par de esquemas, que se hallan en la 4FN, elimina la redundancia que se encontró anteriormente.

Como ocurría cuando se trataba solamente con las dependencias funcionales, resultan interesantes las descomposiciones que carecen de pérdidas y que conservan las dependencias. La siguiente circunstancia relativa a las dependencias multivaloradas y a la ausencia de pérdidas muestra que el algoritmo de la Figura 7.17 sólo genera descomposiciones sin pérdidas:

- Sean R un esquema de relación y D un conjunto de dependencias funcionales y multivaloradas de R . Supóngase que R_1 y R_2 forman una descomposición de R . Esa descomposición de R carecerá de pérdidas si, y sólo si, como mínimo, una de las siguientes dependencias multivaloradas se halla en D^+ :

$$\begin{aligned} R_1 \cap R_2 &\rightarrow\!\!\! \rightarrow R_1 \\ R_1 \cap R_2 &\rightarrow\!\!\! \rightarrow R_2 \end{aligned}$$

Recuérdese que se afirmó en el Apartado 7.4.4 que, si $R_1 \cap R_2 \rightarrow R_1$ o $R_1 \cap R_2 \rightarrow R_2$, entonces R_1 y R_2 son descomposiciones de R sin pérdidas. Esta circunstancia relativa a las dependencias multivaloradas es una declaración más general sobre la carencia de pérdidas. Afirma que, para *toda* descomposición de R sin pérdidas en dos esquemas R_1 y R_2 , debe cumplirse una de las dos dependencias $R_1 \cap R_2 \rightarrow\rightarrow R_1$ o $R_1 \cap R_2 \rightarrow\rightarrow R_2$.

El problema de la conservación de las dependencias al descomponer relaciones se vuelve más complejo en presencia de las dependencias multivaloradas. El Apartado C.1.2 del apéndice aborda este asunto.

7.7 Más formas normales

La cuarta forma normal no es, de ningún modo, la forma normal “definitiva”. Como ya se ha visto, las dependencias multivaloradas ayudan a comprender y a abordar algunas formas de repetición de la información que no pueden comprenderse en términos de las dependencias funcionales. Hay restricciones denominadas **dependencias de reunión** que generalizan las dependencias multivaloradas y llevan a otra forma normal denominada **forma normal de reunión por proyección (FNRP)** (la FNRP se denomina en algunos libros **quinta forma normal**). Hay una clase de restricciones todavía más generales, que lleva a una forma normal denominada **forma normal de dominios y claves (FNDC)**.

Un problema práctico del empleo de estas restricciones generalizadas es que no sólo es difícil razonar con ellas, sino que tampoco hay un conjunto de reglas de inferencia seguras y completas para razonar sobre las restricciones. Por tanto, FNRP y FNDC se utilizan muy rara vez. El Apéndice C ofrece más detalles sobre estas formas normales.

Destaca por su ausencia en este estudio de las formas normales la **segunda forma normal (2FN)**. No se ha estudiado porque sólo es de interés histórico. Simplemente se definirá, y se permitirá al lector experimentar con ella, en el Ejercicio práctico 7.15.

7.8 Proceso de diseño de las bases de datos

Hasta ahora se han examinado aspectos detallados de las formas normales y de la normalización. En este apartado se estudiará el modo de encajar la normalización en el proceso global de diseño de las bases de datos.

Anteriormente, en este capítulo, a partir del Apartado 7.3, se ha dado por supuesto que se tenía un esquema de relación R y que se procedía a normalizarlo. Hay varios modos de obtener ese esquema R :

1. R puede haberse generado al convertir un diagrama E-R en un conjunto de esquemas de relación.
2. R puede haber sido una sola relación que contenía *todos* los atributos que resultaban de interés. El proceso de normalización divide a R en relaciones más pequeñas.
3. R puede haber sido el resultado de algún diseño ad hoc de las relaciones, que hay que comprobar para asegurarse de que satisface la forma normal deseada.

En el resto de este apartado se examinarán las implicaciones de estos enfoques. También se examinarán algunos aspectos prácticos del diseño de las bases de datos, incluida la desnormalización para el rendimiento y ejemplos de mal diseño que no son detectados por la normalización.

7.8.1 El modelo ER y la normalización

Cuando se definen con cuidado los diagramas E-R, identificando correctamente todas las entidades, los esquemas de relación generados a partir de ellos no deben necesitar mucha más normalización. No obstante, puede haber dependencias funcionales entre los atributos de alguna entidad. Por ejemplo, supóngase que la entidad *empleado* tiene los atributos *número_departamento* y *dirección_departamento*, y que hay una dependencia funcional $\text{número_departamento} \rightarrow\rightarrow \text{dirección_departamento}$. Habrá que normalizar la relación generada a partir de *empleado*.

La mayor parte de los ejemplos de este tipo de dependencias surge de un mal diseño del diagrama E-R. En el ejemplo anterior, si se hubiera diseñado correctamente el diagrama E-R, se habría creado una entidad *departamento* con el atributo *dirección_departamento* y una relación entre *empleado* y *departamento*.

De manera parecida, puede que una relación que implique a más de dos entidades no se halle en la forma normal deseable. Como la mayor parte de las relaciones son binarias, estos casos resultan relativamente raros (de hecho, algunas variantes de los diagramas E-R hacen realmente difícil o imposible especificar relaciones no binarias).

Las dependencias funcionales pueden ayudar a detectar un mal diseño E-R. Si las relaciones generadas no se hallan en la forma normal deseada, el problema puede solucionarse en el diagrama E-R. Es decir, la normalización puede llevarse a cabo formalmente como parte del modelado de los datos. De manera alternativa, la normalización puede dejarse a la intuición del diseñador durante el modelado E-R, y puede hacerse formalmente sobre las relaciones generadas a partir del modelo E-R.

El lector atento se habrá dado cuenta de que para que se pudiera ilustrar la necesidad de las dependencias multivaloradas y de la cuarta forma normal hubo que comenzar con esquemas que no se obtuvieron a partir del diseño E-R. En realidad, el proceso de creación de diseños E-R tiende a generar diseños 4FN. Si se cumple alguna dependencia multivalorada y no la implica la dependencia funcional correspondiente, suele proceder de alguna de las fuentes siguientes:

- Una relación de varios a varios.
- Un atributo multivalorado de un conjunto de entidades.

En las relaciones de varios a varios cada conjunto de entidades relacionado tiene su propio esquema y hay un esquema adicional para el conjunto de relaciones. Para los atributos multivalorados se crea un esquema diferente que consta de ese atributo y de la clave primaria del conjunto de entidades (como en el caso del atributo *nombre_subordinado* del conjunto de entidades *empleado*).

El enfoque de las relaciones universales para el diseño de bases de datos relacionales parte de la suposición de que sólo hay un esquema de relación que contenga todos los atributos de interés. Este esquema único define la manera en que los usuarios y las aplicaciones interactúan con la base de datos.

7.8.2 Denominación de los atributos y de las relaciones

Una característica deseable del diseño de bases de datos es la **asunción de un rol único**, lo que significa que cada nombre de atributo tiene un significado único en toda la base de datos. Esto evita que se utilice el mismo atributo para indicar cosas diferentes en esquemas diferentes. Por ejemplo, puede que, de otra manera, se considerara el empleo del atributo *número* para el número de préstamo en el esquema *préstamo* y para el número de cuenta en el esquema *cuenta*. La reunión de una relación del esquema *préstamo* con otra de *cuenta* carece de significado (“información de los pares préstamo-cuenta en los que coincide el número de préstamo y el de cuenta”). Aunque los usuarios y los desarrolladores de aplicaciones pueden trabajar con esmero para garantizar el empleo del *número* correcto en cada circunstancia, tener nombres de atributo diferentes para el número de préstamo y para el de cuenta sirve para reducir los errores de los usuarios. De hecho, se ha observado la suposición de un rol único en los diseños de bases de datos de este libro, y su aplicación es una buena práctica general que conviene seguir.

Aunque es una buena idea hacer que los nombres de los atributos incompatibles sean diferentes, si los atributos de relaciones diferentes tienen el mismo significado, puede ser conveniente emplear el mismo nombre de atributo. Por ejemplo, se han utilizado los nombres de atributo *id_cliente* e *id_empleado* en los conjuntos de entidades *cliente* y *empleado* (y en sus relaciones). Si deseáramos generalizar esos conjuntos de entidades mediante la creación del conjunto de entidades *persona*, habría que renombrar el atributo. Por tanto, aunque no se tenga actualmente una generalización de *cliente* y de *empleado*, si se prevé esa posibilidad es mejor emplear el mismo nombre en los dos conjuntos de relaciones (y en sus relaciones).

Aunque, técnicamente, el orden de los nombres de los atributos en los esquemas no tiene ninguna importancia, es costumbre relacionar en primer lugar los atributos de la clave primaria. Esto facilita la lectura de los resultados predeterminados (como los generados por *select* *).

En los esquemas de bases de datos de gran tamaño los conjuntos de relaciones (y los esquemas derivados) se suelen denominar mediante la concatenación de los nombres de los conjuntos de entidades a los que hacen referencia, quizás con guiones o caracteres de subrayado intercalados. En este libro se han utilizado algunos nombres de este tipo, por ejemplo, *sucursal_cuenta* y *sucursal_préstamo*. Se han utilizado los nombres *prestatario* o *impositor* en lugar de otros concatenados de mayor longitud como *cliente*

_préstamo o *cliente_cuenta*. Esto resultaba aceptable, ya que no es difícil recordar las entidades asociadas a unas pocas relaciones. No siempre se pueden crear nombres de relaciones mediante la mera concatenación; por ejemplo, la relación jefe o trabaja-para entre empleados no tendría mucho sentido si se llamara *empleado_empleado*. De manera parecida, si hay varios conjuntos de relaciones posibles entre un par de conjuntos de entidades, los nombres de las relaciones deben incluir componentes adicionales para identificar cada relación.

Las diferentes organizaciones tienen costumbres diferentes para la denominación de las entidades. Por ejemplo, a un conjunto de entidades de clientes se le puede denominar *cliente* o *clientes*. En los diseños de las bases de datos de este libro se ha decidido utilizar la forma singular. Es aceptable tanto el empleo del singular como el del plural, siempre y cuando la convención se utilice de manera consistente en todas las entidades.

A medida que los esquemas aumentan de tamaño, con un número creciente de relaciones, el empleo de una denominación consistente de los atributos, de las relaciones y de las entidades facilita mucho la vida de los diseñadores de bases de datos y de los programadores de aplicaciones.

7.8.3 Desnormalización para el rendimiento

A veces, los diseñadores de bases de datos escogen un esquema que tiene información redundante; es decir, que no está normalizado. Utilizan la redundancia para mejorar el rendimiento de aplicaciones concretas. La penalización sufrida por no emplear un esquema normalizado es el trabajo adicional (en términos de tiempos de codificación y de ejecución) de mantener consistentes los datos redundantes.

Por ejemplo, supóngase que hay que mostrar el nombre del titular junto con el número de cuenta y con el saldo cada vez que se accede a la cuenta. En el esquema normalizado esto exige una reunión de *cuenta* con *impositor*.

Una alternativa al cálculo de la reunión sobre la marcha es almacenar una relación que contenga todos los atributos de *cuenta* y de *impositor*. Esto hace más rápida la visualización de la información de la cuenta. Sin embargo, la información del saldo de la cuenta se repite para cada uno de los titulares, y la aplicación debe actualizar todas las copias cada vez que se actualice el saldo. El proceso de tomar un esquema normalizado y hacer que no esté normalizado se denomina **desnormalización**, y los diseñadores lo utilizan para ajustar el rendimiento de los sistemas para que den soporte a las operaciones críticas en el tiempo.

Una opción mejor, soportada hoy en día por muchos sistemas de bases de datos, es emplear el esquema normalizado y, además, almacenar la reunión de *cuenta* e *impositor* en forma de vista materializada (recuérdese que las vistas materializadas son vistas cuyo resultado se almacena en la base de datos y se actualiza cuando se actualizan las relaciones utilizadas en ellas). Al igual que la desnormalización, el empleo de las vistas materializadas supone sobrecargas de espacio y de tiempo; sin embargo, presenta la ventaja de que conservar actualizadas las vistas es labor del sistema de bases de datos, no del programador de la aplicación.

7.8.4 Otros problemas del diseño

Hay algunos aspectos del diseño de bases de datos que la normalización no aborda y, por tanto, pueden llevar a un mal diseño de la base de datos. Los datos relativos al tiempo o a intervalos temporales presentan varios de esos problemas. A continuación se ofrecen algunos ejemplos; evidentemente, conviene evitar esos diseños.

Considérese una base de datos empresarial, en la que se desea almacenar los beneficios de varias compañías a lo largo de varios años. Se puede utilizar la relación *beneficios* (*id_empresa*, *año*, *importe*) para almacenar la información sobre los beneficios. La única dependencia funcional de esta relación es *id_empresa, año → importe*, y se halla en la FNBC.

Un diseño alternativo es el empleo de varias relaciones, cada una de las cuales almacena los beneficios de un año diferente. Supóngase que los años que nos interesan son 2000, 2001 y 2002; se tendrán, entonces, relaciones de la forma *beneficios_2000*, *beneficios_2001* y *beneficios_2002*, todas las cuales se hallan en el esquema (*id_empresa*, *beneficios*). En este caso, la única dependencia funcional de cada relación será *id_empresa → beneficios*, por lo que esas relaciones también se hallan en la FNBC.

No obstante, este diseño alternativo es, claramente, una mala idea—habría que crear una relación nueva cada año, y también habría que escribir consultas nuevas todos los años, para tener en cuenta cada nueva relación. Las consultas también serían más complicadas, ya que probablemente tendrían que hacer referencia a muchas relaciones.

Otra manera de representar esos mismos datos es tener una sola relación *año_empresa* (*id_empresa*, *beneficios_2000*, *beneficios_2001*, *beneficios_2002*). En este caso, las únicas dependencias funcionales van de *id_empresa* hacia los demás atributos y, una vez más, la relación se halla en la FNBC. Este diseño también es una mala idea, ya que tiene problemas parecidos a los del diseño anterior—es decir, habría que modificar el esquema de la relación y escribir consultas nuevas cada año. Las consultas también serían más complicadas, ya que puede que tuvieran que hacer referencia a muchos atributos.

Las representaciones como las de la relación *año_empresa*, con una columna para cada valor de cada atributo, se denominan de **referencias cruzadas**; se emplean mucho en las hojas de cálculo, en los informes y en las herramientas de análisis de datos. Aunque esas representaciones resultan útiles para mostrárselas a los usuarios, por las razones que se acaban de dar, no resultan deseables en el diseño de bases de datos. Se han propuesto extensiones del SQL para pasar los datos de la representación relacional normal a la de referencias cruzadas, para su visualización.

7.9 Modelado de datos temporales

Supóngase que en el banco se conservan datos que no sólo muestran la dirección actual de cada cliente, sino también todas las direcciones anteriores de las que el banco tenga noticia. Se pueden formular consultas como “Averiguar todos los clientes que vivían en Vigo en 1981”. En ese caso, puede que se tengan varias direcciones por cliente. Cada dirección tiene asociadas una fecha de comienzo y otra de finalización, que indican el periodo en que el cliente residió en esa dirección. Se puede utilizar un valor especial para la fecha de finalización, por ejemplo, nulo, o un valor que se halle claramente en el futuro, como 31/12/9999, para indicar que el cliente sigue residiendo en esa dirección.

En general, los **datos temporales** son datos que tienen asociado un intervalo de tiempo durante el cual son **válidos**⁴. Se utiliza el término **instantánea** de los datos para indicar su valor en un momento determinado. Por tanto, una instantánea de los datos de los clientes muestra el valor de todos los atributos de los clientes, como la dirección, en un momento concreto.

El modelado de datos temporales es una cuestión interesante por varios motivos. Por ejemplo, supóngase que se tiene una entidad *cliente* con la que se desea asociar una dirección que varía con el tiempo. Para añadir información temporal a una dirección hay que crear un atributo multivalorado, cada uno de cuyos valores es un valor compuesto que contiene una dirección y un intervalo de tiempo. Además de los valores de los atributos que varían con el tiempo, puede que las propias entidades tengan un periodo de validez asociado. Por ejemplo, la entidad cuenta puede tener un periodo de validez desde la fecha de apertura hasta la de cancelación. Las relaciones también pueden tener asociados periodos de validez. Por ejemplo, la relación *impositor* entre un cliente y una cuenta puede registrar el momento en que el cliente pasó a ser titular de la cuenta. Por tanto, habría que añadir intervalos de validez a los valores de los atributos, a las entidades y a las relaciones. La adición de esos detalles a los diagramas E-R hace que sean muy difíciles de crear y de comprender. Ha habido varias propuestas para extender la notación E-R para que especifique de manera sencilla que un atributo o una relación varía con el tiempo, pero no hay ninguna norma aceptada al respecto.

Cuando se realiza un seguimiento del valor de los datos a lo largo del tiempo, puede que dejen de cumplirse dependencias funcionales que se suponía que se cumplían, como

$$id_cliente \rightarrow calle_cliente\ ciudad_cliente$$

En su lugar, se cumpliría la restricción siguiente (expresada en castellano): “*id_cliente* sólo tiene un valor de *calle_cliente* y de *ciudad_cliente* para cada momento *t* dado”.

Las dependencias funcionales que se cumplen en un momento concreto se denominan dependencias funcionales temporales. Formalmente, la **dependencia funcional temporal** $X \xrightarrow{\tau} Y$ se cumple en el

4. Hay otros modelos de datos temporales que distinguen entre **periodo de validez** y **momento de la transacción**; el último registra el momento en que se registró un hecho en la base de datos. Para simplificar se prescinde de estos detalles.

esquema de relación R si, para todos los ejemplares legales r de R , todas las instantáneas de r satisfacen la dependencia funcional $X \rightarrow Y$.

Se puede extender la teoría del diseño de bases de datos relacionales para que tenga en cuenta las dependencias funcionales temporales. Sin embargo, el razonamiento con las dependencias funcionales normales ya resulta bastante difícil, y pocos diseñadores están preparados para trabajar con las dependencias funcionales temporales.

En la práctica, los diseñadores de bases de datos recurren a enfoques más sencillos para diseñar las bases de datos temporales. Un enfoque empleado con frecuencia es diseñar toda la base de datos (incluidos los diseños E-R y relacional) ignorando las modificaciones temporales (o, lo que es lo mismo, tomando sólo una instantánea en consideración). Tras esto, el diseñador estudia las diferentes relaciones y decide las que necesitan que se realice un seguimiento de su variación temporal.

El paso siguiente es añadir información sobre los períodos de validez a cada una de esas relaciones, añadiendo como atributos el momento de inicio y el de finalización. Por ejemplo, supóngase que se tiene la relación

$$\text{asignatura} (id_asignatura, denominación_asignatura)$$

que asocia la denominación de cada asignatura con la asignatura correspondiente, que queda identificada mediante su identificador de asignatura. La denominación de la asignatura puede variar con el tiempo, lo cual se puede tener en cuenta añadiendo un rango de validez; el esquema resultante sería:

$$\text{asignatura} (id_asignatura, denominación_asignatura, comienzo, final)$$

Un ejemplar de esta relación puede tener dos registros (CS101, “Introducción a la programación”, 01/01/1985, 31/12/2000) y (CS101, “Introducción a C”, 01/01/2001, 31/12/9999). Si se actualiza la relación para cambiar la denominación de la asignatura a “Introducción a Java”, se actualizaría la fecha “31/12/9999” a la correspondiente al momento hasta el que fue válido el valor anterior (“Introducción a C”), y se añadiría una tupla nueva que contendría la nueva denominación (“Introducción a Java”), con la fecha de inicio correspondiente.

Si otra relación tuviera una clave externa que hiciera referencia a una relación temporal, el diseñador de la base de datos tendría que decidir si la referencia se hace a la versión actual de los datos o a los datos de un momento concreto. Por ejemplo, una relación que registre la asignación de aulas actual para cada asignatura puede hacer referencia de manera implícita al valor temporal actual asociado con cada $id_asignatura$. Por otro lado, los registros del expediente de cada estudiante deben hacer referencia a la denominación de la asignatura en el momento en que la cursó ese estudiante. En este último caso, la relación que hace la referencia también debe almacenar la información temporal, para identificar cada registro concreto de la relación $asignatura$.

La clave primaria original de una relación temporal ya no identificaría de manera única a cada tupla. Para solucionar este problema se pueden añadir a la clave primaria los atributos de fecha de inicio y de finalización. Sin embargo, persisten algunos problemas:

- Es posible almacenar datos con intervalos que se solapen, pero la restricción de clave primaria no puede detectarlo. Si el sistema soporta un tipo nativo *periodo de validez*, puede detectar y evitar esos intervalos temporales que se solapan.
- Para especificar una clave externa que haga referencia a una relación así, las tuplas que hacen la referencia tienen que incluir los atributos de momento inicial y final como parte de su clave externa, y los valores deberán coincidir con los de la tupla a la que hacen referencia. Además, si la tupla a la que hacen referencia se actualiza (y el momento final que se hallaba en el futuro se actualiza), esa actualización debe propagarse a todas las tuplas que hacen referencia a ella.

Si el sistema soporta los datos temporales de alguna manera mejor, se puede permitir que la tupla que hace la referencia especifique un momento, en lugar de un rango temporal, y confiar en que el sistema garantice que hay una tupla en la relación a la que hace referencia cuyo periodo de validez contenga ese momento. Por ejemplo, un registro de un expediente puede especificar $id_asignatura$ y un momento (por ejemplo, la fecha de comienzo de un trimestre), lo que basta para identificar el registro correcto de la relación $asignatura$.

Como caso especial frecuente, si todas las referencias a los datos temporales se hacen exclusivamente a los datos actuales, una solución más sencilla es no añadir información temporal a la relación y, en su lugar, crear la relación *historial* correspondiente que contenga esa información temporal, para los valores del pasado. Por ejemplo, en la base de datos bancaria, se puede utilizar el diseño que se ha creado, ignorando las modificaciones temporales, para almacenar únicamente la información actual. Toda la información histórica se traslada a las relaciones históricas. Por tanto, la relación *cliente* sólo puede almacenar la dirección actual, mientras que la relación *historial_cliente* puede contener todos los atributos de *cliente*, con los atributos adicionales *momento_inicial* y *momento_final*.

Aunque no se ha ofrecido ningún método formal para tratar los datos temporales, los problemas estudiados y los ejemplos ofrecidos deben ayudar al lector a diseñar bases de datos que registren datos temporales. Más adelante, en el Apartado 24.2, se tratan otros aspectos del manejo de datos temporales, incluidas las consultas temporales.

7.10 Resumen

- Se han mostrado algunas dificultades del diseño de bases de datos y el modo de diseñar de manera sistemática esquemas de bases de datos que eviten esas dificultades. Entre esas dificultades están la información repetida y la imposibilidad de representar cierta información.
- Se ha mostrado el desarrollo del diseño de bases de datos relacionales a partir de los diseños E-R, cuándo los esquemas se pueden combinar con seguridad y cuándo se deben descomponer. Todas las descomposiciones válidas deben ser sin pérdidas.
- Se han descrito las suposiciones de dominios atómicos y de primera forma normal.
- Se ha introducido el concepto de las dependencias funcionales y se ha utilizado para presentar dos formas normales, la forma normal de Boyce–Codd (FNBC) y la tercera forma normal (3NF).
- Si la descomposición conserva las dependencias, dada una actualización de la base de datos, todas las dependencias funcionales pueden verificarse a partir de las diferentes relaciones, sin necesidad de calcular la reunión de las relaciones de la descomposición.
- Se ha mostrado la manera de razonar con las dependencias funcionales. Se ha puesto un énfasis especial en señalar las dependencias que están implicadas lógicamente por conjuntos de dependencias. También se ha definido el concepto de recubrimiento canónico, que es un conjunto mínimo de dependencias funcionales equivalente a un conjunto dado de dependencias funcionales.
- Se ha descrito un algoritmo para la descomposición de las relaciones en la FNBC. Hay relaciones para las cuales no hay ninguna descomposición en la FNBC que conserve las dependencias.
- Se han utilizado los recubrimientos canónicos para descomponer las relaciones en la 3NF, que es una pequeña relajación de las condiciones de la FNBC. Las relaciones en la 3NF pueden tener alguna redundancia, pero siempre hay una descomposición en la 3NF que conserva las dependencias.
- Se ha presentado el concepto de dependencias multivaloradas, que especifican las restricciones que no pueden especificarse únicamente con las dependencias funcionales. Se ha definido la cuarta forma normal (4FN) con las dependencias multivaloradas. El Apartado C.1.1 del apéndice da detalles del razonamiento sobre las dependencias multivaloradas.
- Otras formas normales, como la FNRP y la FNDC, eliminan formas más sutiles de redundancia. Sin embargo, es difícil trabajar con ellas y se emplean rara vez. El Apéndice C ofrece detalles de estas formas normales.
- Al revisar los temas de este capítulo hay que tener en cuenta que el motivo de que se hayan podido definir enfoques rigurosos del diseño de bases de datos relacionales es que el modelo relacional de datos descansa sobre una base matemática sólida. Ésa es una de las principales ventajas del modelo relacional en comparación con los otros modelos de datos que se han estudiado.

Términos de repaso

- Modelo E-R y normalización.
- Descomposición.
- Dependencias funcionales.
- Descomposición sin pérdidas.
- Dominios atómicos.
- Primera forma normal (1FN).
- Relaciones legales.
- Superclave.
- R satisface F .
- F se cumple en R .
- Forma normal de Boyce–Codd (FNBC).
- Conservación de las dependencias.
- Tercera forma normal (3NF).
- Dependencias funcionales triviales.
- Cierre de un conjunto de dependencias funcionales.
- Axiomas de Armstrong.
- Cierre de los conjuntos de atributos.
- Restricción de F a R_i .
- Recubrimiento canónico.
- Atributos raros.
- Algoritmo de descomposición en la FNBC.
- Algoritmo de descomposición en la 3NF.
- Dependencias multivaloradas.
- Cuarta forma normal (4FN).
- Restricción de las dependencias multivaloradas.
- Forma normal de reunión por proyección (FNRP).
- Forma normal de dominios y claves (FNDC).
- Relación universal.
- Suposición de un rol único.
- Desnormalización.

Ejercicios prácticos

7.1 Supóngase que se descompone el esquema $R = (A, B, C, D, E)$ en

$$\begin{array}{l} (A, B, C) \\ (A, D, E) \end{array}$$

Demuéstrese que esta descomposición es una descomposición sin pérdidas si se cumple el siguiente conjunto F de dependencias funcionales:

$$\begin{array}{l} A \rightarrow BC \\ CD \rightarrow E \\ B \rightarrow D \\ E \rightarrow A \end{array}$$

7.2 Indíquense todas las dependencias funcionales que satisface la relación de la Figura 7.18.

7.3 Explíquese el modo en que se pueden utilizar las dependencias funcionales para indicar:

- Existe un conjunto de relaciones de uno a uno entre los conjuntos de entidades *cuenta* y *cliente*.
- Existe un conjunto de relaciones de varios a uno entre los conjuntos de entidades *cuenta* y *cliente*.

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Figura 7.18 La relación del Ejercicio práctico 7.2.

- 7.4** Empléense los axiomas de Armstrong para probar la corrección de la regla de la unión. *Sugerencia:* utilícese la regla de la aumentatividad para probar que, si $\alpha \rightarrow \beta$, entonces $\alpha \rightarrow \alpha\beta$. Aplíquese nuevamente la regla de la aumentatividad, utilizando $\alpha \rightarrow \gamma$, y aplíquese luego la regla de la transitividad.
- 7.5** Empléense los axiomas de Armstrong para probar la corrección de la regla de la pseudotransitividad.
- 7.6** Calcúlese el cierre del siguiente conjunto F de relaciones funcionales para el esquema de relación $R = (A, B, C, D, E)$.

$$\begin{aligned} A &\rightarrow BC \\ CD &\rightarrow E \\ B &\rightarrow D \\ E &\rightarrow A \end{aligned}$$

Indíquense las claves candidatas de R .

- 7.7** Utilizando las dependencias funcionales del Ejercicio práctico 7.6, calcúlese el recubrimiento canónico F_c .
- 7.8** Considérese el algoritmo de la Figura 7.19 para calcular α^+ . Demuéstrese que este algoritmo resulta más eficiente que el presentado en la Figura 7.9 (Apartado 7.4.2) y que calcula α^+ de manera correcta.
- 7.9** Dado el esquema de base de datos $R(a, b, c)$ y una relación r del esquema R , escríbase una consulta SQL para comprobar si la dependencia funcional $b \rightarrow c$ se cumple en la relación r . Escríbase también un aserto de SQL que haga que se cumpla la dependencia funcional. Supóngase que no hay ningún valor nulo.
- 7.10** Sea R_1, R_2, \dots, R_n una descomposición del esquema U . Sea $u(U)$ una relación y sea $r_i = \Pi_{R_i}(u)$. Demuéstrese que

$$u \subseteq r_1 \bowtie r_2 \bowtie \dots \bowtie r_n$$

- 7.11** Demuéstrese que la descomposición del Ejercicio práctico 7.1 no es una descomposición que conserve las dependencias.
- 7.12** Demuéstrese que es posible asegurar que una descomposición que conserve las dependencias en la 3FN sea una descomposición sin pérdidas garantizando que, como mínimo, un esquema contenga una clave candidata para el esquema que se está descomponiendo. *Sugerencia:* demuéstrese que la reunión de todas las proyecciones en los esquemas de la descomposición no puede tener más tuplas que la relación original.
- 7.13** Dese un ejemplo de esquema de relación R' y de un conjunto F' de dependencias funcionales tales que haya, al menos, tres descomposiciones sin pérdidas distintas de R' en FNBC.
- 7.14** Sea atributo *primo* el que aparece, como mínimo, en una clave candidata. Sean α y β conjuntos de atributos tales que se cumple $\alpha \rightarrow \beta$, pero no se cumple $\beta \rightarrow \alpha$. Sea A un atributo que no esté en α ni en β y para el que se cumpla que $\beta \rightarrow A$. Se dice que A es *dependiente de manera transitiva* de α . Se puede reformular la definición de la 3FN de la manera siguiente: el esquema de relación R está en la 3FN con respecto al conjunto F de dependencias funcionales si no hay atributos no primos A en R para los cuales A sea dependiente de manera transitiva de una clave de R .

Demuéstrese que esta nueva definición es equivalente a la original.

- 7.15** La dependencia funcional $\alpha \rightarrow \beta$ se denomina **dependencia parcial** si hay un subconjunto propio γ de α tal que $\gamma \rightarrow \beta$. Se dice que β es *parcialmente dependiente* de α . El esquema de relación R está en la **segunda forma normal** (2FN) si cada atributo A de R cumple uno de los criterios siguientes:
- Aparece en una clave candidata.

```

resultado := ∅;
/* cuentadf es un array cuyo elemento  $i$ -ésimo contiene
   el número de atributos del lado izquierdo de la  $i$ -ésima
   DF que todavía no se sabe que estén en  $\alpha^+$  */
for  $i := 1$  to  $|F|$  do
begin
  Supóngase que  $\beta \rightarrow \gamma$  denota la  $i$ -ésima DF;
  cuentadf [ $i$ ] :=  $|\beta|$ ;
end
/* aparece es un array con una entrada por cada atributo. La
   entrada del atributo  $A$  es una lista de enteros. Cada entero
    $i$  de la lista indica que  $A$  aparece en el lado izquierdo de
   la  $i$ -ésima DF */
for each atributo  $A$  do
begin
  aparece [ $A$ ] := NIL;
  for  $i := 1$  to  $|F|$  do
begin
  Supóngase que  $\beta \rightarrow \gamma$  denota la  $i$ -ésima DF;
  if  $A \in \beta$  then añadir  $i$  a aparece [ $A$ ];
end
end
agregar ( $\alpha$ );
return (resultado);

procedure agregar ( $\alpha$ );
for each atributo  $A$  in  $\alpha$  do
begin
  if  $A \notin resultado$  then
begin
  resultado := resultado  $\cup \{A\}$ ;
  for each elemento  $i$  in aparece [ $A$ ] do
begin
  cuentadf [ $i$ ] := cuentadf [ $i$ ] - 1;
  if cuentadf [ $i$ ] := 0 then
begin
  supóngase que  $\beta \rightarrow \gamma$  denota la  $i$ -ésima FD;
  agregar ( $\gamma$ );
end
end
end
end
end

```

Figura 7.19 Algoritmo para calcular α^+ .

- No es parcialmente dependiente de una clave candidata.

Demuéstrese que cada esquema en la 3FN se halla en la 2FN. *Sugerencia:* demuéstrese que todas las dependencias parciales son dependencias transitivas.

7.16 Dese un ejemplo de esquema de relación R y un conjunto de dependencias tales que R se halle en la FNBC, pero no en la 4FN.

Ejercicios

- 7.17 Explíquese lo que se quiere decir con *repetición de la información e imposibilidad de representación de la información*. Explíquese el motivo por el que estas propiedades pueden indicar un mal diseño de las bases de datos relacionales.
- 7.18 Indíquese el motivo de que ciertas dependencias funcionales se denominen dependencias funcionales *triviales*.
- 7.19 Utilícese la definición de dependencia funcional para argumentar que cada uno de los axiomas de Armstrong (reflexividad, aumentatividad y transitividad) es correcto.
- 7.20 Considérese la siguiente regla propuesta para las dependencias funcionales: si $\alpha \rightarrow \beta$ y $\gamma \rightarrow \beta$, entonces $\alpha \rightarrow \gamma$. Pruébese que esta regla *no* es correcta mostrando una relación r que satisfaga $\alpha \rightarrow \beta$ y $\gamma \rightarrow \beta$, pero no $\alpha \rightarrow \gamma$.
- 7.21 Utilíicense los axiomas de Armstrong para probar la corrección de la regla de la descomposición.
- 7.22 Utilizando las dependencias funcionales del Ejercicio práctico 7.6, calcúlese B^+ .
- 7.23 Demuéstrese que la siguiente descomposición del esquema R del Ejercicio práctico 7.1 no es una descomposición sin pérdidas:

$$\begin{array}{l} (A, B, C) \\ (C, D, E) \end{array}$$

Sugerencia: dese un ejemplo de una relación r del esquema R tal que

$$\Pi_{A, B, C}(r) \bowtie \Pi_{C, D, E}(r) \neq r$$

- 7.24 Indíquense los tres objetivos de diseño de las bases de datos relacionales y explíquese el motivo de que cada uno de ellos sea deseable.
- 7.25 Dese una descomposición sin pérdidas en la FNBC del esquema R del Ejercicio práctico 7.1.
- 7.26 Al diseñar una base de datos relacional, indíquese el motivo de que se pueda escoger un diseño que no esté en la FNBC.
- 7.27 Dese una descomposición sin pérdidas en la 3FN que conserve las dependencias del esquema R del Ejercicio práctico 7.1.
- 7.28 Dados los tres objetivos del diseño de bases de datos relacionales, indíquese si hay alguna razón para diseñar un esquema de base de datos que se halle en la 2FN, pero que no se halle en ninguna forma normal de orden superior (véase el Ejercicio práctico 7.15 para obtener la definición de la 2FN).
- 7.29 Dado el esquema relacional $r(A, B, C, D)$, ¿implica lógicamente $A \rightarrow\!\!> BC$ a $A \rightarrow\!\!> B$ y a $A \rightarrow\!\!> C$? En caso positivo, pruébese; en caso contrario, dese un contraejemplo.
- 7.30 Explíquese el motivo de que la 4FN sea una forma normal más deseable que la FNBC.

Notas bibliográficas

El primer estudio de la teoría del diseño de bases de datos relacionales apareció en un artículo pionero de Codd [1970]. En ese artículo, Codd introducía también las dependencias funcionales y la primera, la segunda y la tercera formas normales.

Los axiomas de Armstrong se introdujeron en Armstrong [1974]. A finales de los años setenta se produjo un desarrollo significativo de la teoría de las bases de datos relacionales. Esos resultados se recogen en varios textos sobre la teoría de las bases de datos, como Maier [1983], Atzeni y Antonellis [1993] y Abiteboul et al. [1995].

La FNBC se introdujo en Codd [1972]. Biskup et al. [1979] dan el algoritmo que se ha utilizado para encontrar descomposiciones en la 3FN que conserven las dependencias. Los resultados fundamentales de la propiedad de la descomposición sin pérdidas aparecen en Aho et al. [1979a].

Beeri et al. [1977] dan un conjunto de axiomas para las dependencias multivaluadas y prueba que los axiomas de los autores son correctos y completos. Los conceptos de 4FN, de FNRP y de FNDC son de Fagin [1977], Fagin [1979] y de Fagin [1981], respectivamente. Véanse las notas bibliográficas del Apéndice C para tener más referencias sobre la literatura relativa a la normalización.

Jensen et al. [1994] presenta un glosario de conceptos relacionados con las bases de datos temporales. Gregersen y Jensen [1999] presenta un resumen de las extensiones del modelo E-R para el tratamiento de datos temporales. Tansel et al. [1993] trata la teoría, el diseño y la implementación de las bases de datos temporales. Jensen et al. [1996] describe las extensiones de la teoría de la dependencia a los datos temporales.

Bases de datos orientadas a objetos y XML

Varias áreas de aplicaciones para los sistemas de bases de datos se hallan limitadas por las restricciones del modelo de datos relacional. En consecuencia, los investigadores han desarrollado varios modelos de datos basados en enfoques orientados a objetos para tratar con esos dominios de aplicaciones.

El modelo relacional orientado a objetos, que se describe en el Capítulo 9, combina características del modelo relacional y del modelo orientado a objetos. Este modelo proporciona el rico sistema de tipos de los lenguajes orientados a objetos combinado con las relaciones como base del almacenamiento de los datos. Aplica la herencia a las relaciones, no sólo a los tipos. El modelo de datos relacional orientado a objetos permite una migración fácil desde las bases de datos relacionales, lo que resulta atractivo para las diferentes marcas de bases de datos relacionales. En consecuencia, la norma SQL:1999 incluye varias características orientadas a objetos en su sistema de tipos, mientras que sigue utilizando el modelo relacional como modelo subyacente.

El término base de datos orientada a objetos se utiliza para describir los sistemas de bases de datos que soportan el acceso directo a los datos desde lenguajes de programación orientados a objetos, sin necesidad de lenguajes de consultas relacionales como interfaz de las bases de datos. El Capítulo 9 también proporciona una breve visión general de las bases de datos orientadas a objetos.

El lenguaje XML se diseñó inicialmente como un modo de añadir información de marcas a los documentos de texto, pero ha adquirido importancia debido a sus aplicaciones en el intercambio de datos. XML permite representar los datos mediante una estructura anidada y, además, ofrece una gran flexibilidad en su estructuración, lo que resulta importante para ciertos tipos de datos no tradicionales. El Capítulo 10 describe el lenguaje XML y a continuación presenta diferentes formas de expresar las consultas sobre datos representados en XML, incluido el lenguaje de consultas XQuery para XML, que está adquiriendo una gran aceptación.

Bases de datos basadas en objetos

Las aplicaciones tradicionales de las bases de datos consisten en tareas de procesamiento de datos, como la gestión bancaria y de nóminas, con tipos de datos relativamente sencillos, que se adaptan bien al modelo relacional. A medida que los sistemas de bases de datos se fueron aplicando a un rango más amplio de aplicaciones, como el diseño asistido por computadora y los sistemas de información geográfica, las limitaciones impuestas por el modelo relacional se convirtieron en un obstáculo. La solución fue la introducción de bases de datos basadas en objetos, que permiten trabajar con tipos de datos complejos.

9.1 Visión general

El primer obstáculo al que se enfrentan los programadores que usan el modelo relacional de datos es el limitado sistema de tipos soportado por el modelo relacional. Los dominios de aplicación complejos necesitan tipos de datos del mismo nivel de complejidad, como las estructuras de registros anidados, los atributos multivalorados y la herencia, que los lenguajes de programación tradicionales soportan. La notación E-R y las notaciones E-R extendidas soportan, de hecho, estas características, pero hay que trasladarlas a tipos de datos de SQL más sencillos. El **modelo de datos relacional orientado a objetos** extiende el modelo de datos relacional ofreciendo un sistema de tipos más rico que incluye tipos de datos complejos y orientación a objetos. Hay que extender de manera acorde los lenguajes de consultas relacionales, en especial SQL, para que puedan trabajar con este sistema de tipos más rico. Estas extensiones intentan conservar los fundamentos relacionales—en especial, el acceso declarativo a los datos—mientras extienden la potencia de modelado. Los **sistemas de bases de datos relacionales basadas en objetos**, es decir, los sistemas de bases de datos basados en el modelo objeto-relación, ofrecen un medio de migración cómodo para los usuarios de las bases de datos relacionales que deseen usar características orientadas a objetos.

El segundo obstáculo es la dificultad de acceso a los datos de la base de datos desde los programas escritos en lenguajes de programación como C++ o Java. La mera extensión del sistema de tipos soportado por las bases de datos no resulta suficiente para resolver completamente este problema. Las diferencias entre el sistema de tipos de las bases de datos y el de los lenguajes de programación hace más complicados el almacenamiento y la recuperación de los datos, y se debe minimizar. Tener que expresar el acceso a las bases de datos mediante un lenguaje (SQL) que es diferente del lenguaje de programación también hace más difícil el trabajo del programador. Es deseable, para muchas aplicaciones, contar con estructuras o extensiones del lenguaje de programación que permitan el acceso directo a los datos de la base de datos, sin tener que pasar por un lenguaje intermedio como SQL.

El término **lenguajes de programación persistentes** hace referencia a las extensiones de los lenguajes de programación existentes que añaden persistencia y otras características de las bases de datos usando el sistema de tipos nativo del lenguaje de programación. El término **sistemas de bases de datos orientadas a objetos** se usa para hacer referencia a los sistemas de bases de datos que soportan sistemas de

tipos orientados a objetos y permiten el acceso directo a los datos desde los lenguajes de programación orientados a objetos usando el sistema de tipos nativo del lenguaje.

En este capítulo se explicará primero el motivo del desarrollo de los tipos de datos complejos. Luego se estudiarán los sistemas de bases de datos relacionales orientados a objetos; el tratamiento se basará en las extensiones relacionales orientadas a objetos añadidas a la versión SQL:1999 de la norma de SQL. La descripción se basa en la norma de SQL, concretamente, en el uso de características que se introdujeron en SQL:1999 y en SQL:2003. Téngase en cuenta que la mayor parte de los productos de bases de datos sólo soportan un subconjunto de las características de SQL aquí descritas. Se debe consultar el manual de usuario del sistema de bases de datos que se utilice para averiguar las características que soporta.

Luego se estudian brevemente los sistemas de bases de datos orientados a objetos que añaden el soporte de la persistencia a los lenguajes de programación orientados a objetos. Finalmente, se describen situaciones en las que el enfoque relacional orientado a objetos es mejor que el enfoque orientado a objetos, y viceversa, y se mencionan criterios para escoger entre los dos.

9.2 Tipos de datos complejos

Las aplicaciones de bases de datos tradicionales consisten en tareas de procesamiento de datos, tales como la banca y la gestión de nóminas. Dichas aplicaciones presentan conceptualmente tipos de datos simples. Los elementos de datos básicos son registros bastante pequeños y cuyos campos son atómicos, es decir, no contienen estructuras adicionales y en los que se cumple la primera forma normal (véase el Capítulo 7). Además, sólo hay unos pocos tipos de registros.

En los últimos años, ha crecido la demanda de formas de abordar tipos de datos más complejos. Considerense, por ejemplo, las direcciones. Mientras que una dirección completa se puede considerar como un elemento de datos atómico del tipo cadena de caracteres, esa forma de verlo esconde detalles como la calle, la población, la provincia, y el código postal, que pueden ser interesantes para las consultas. Por otra parte, si una dirección se representa dividiéndola en sus componentes (calle, población, provincia y código postal) la escritura de las consultas sería más complicada, pues tendrían que mencionar cada campo. Una alternativa mejor es permitir tipos de datos estructurados, que admiten el tipo *dirección* con las subpartes *calle*, *población*, *provincia* y *código_postal*.

Como ejemplo adicional, considérense los atributos multivalorados del modelo E-R. Esos atributos resultan naturales, por ejemplo, para la representación de números de teléfono, ya que las personas pueden tener más de un teléfono. La alternativa de la normalización mediante la creación de una nueva relación resulta costosa y artificial para este ejemplo.

Con sistemas de tipos complejos se pueden representar directamente conceptos del modelo E-R, como los atributos compuestos, los atributos multivalorados, la generalización y la especialización, sin necesidad de una compleja traducción al modelo relacional.

En el Capítulo 7 se definió la *primera forma normal* (1FN), (1FN) que exige que todos los atributos tengan *dominios atómicos*. Recuérdese que un dominio es *atómico* si se considera que los elementos del dominio son unidades indivisibles.

La suposición de la 1FN es natural en los ejemplos bancarios que se han considerado. No obstante, no todas las aplicaciones se modelan mejor mediante relaciones en la 1FN. Por ejemplo, en vez de considerar la base de datos como un conjunto de registros, los usuarios de ciertas aplicaciones la ven como un conjunto de objetos (o de entidades). Puede que cada objeto necesite varios registros para su representación. Una interfaz sencilla y fácil de usar necesita una correspondencia de uno a uno entre el concepto intuitivo de objeto del usuario y el concepto de elemento de datos de la base de datos.

Considérese, por ejemplo, una aplicación para una biblioteca y supóngase que se desea almacenar la información siguiente para cada libro:

- Título del libro
- Lista de autores
- Editor
- Conjunto de palabras clave

Es evidente que, si se define una relación para esta información varios dominios no son atómicos.

- **Autores.** Cada libro puede tener una lista de autores, que se pueden representar como array. Pese a todo, puede que se desee averiguar todos los libros de los que es autor Santos. Por tanto, lo que interesa es una subparte del elemento del dominio “autores”.
- **Palabras clave.** Si se almacena un conjunto de palabras clave para cada libro, se espera poder recuperar todos los libros cuyas palabras clave incluyan uno o más términos dados. Por tanto, se considera el dominio del conjunto de palabras clave como no atómico.
- **Editor.** A diferencia de *palabras clave* y *autores*, *editor* no tiene un dominio que se evalúe en forma de conjunto. No obstante, se puede considerar que *editor* consta de los subcampos *nombre* y *sucursal*. Este punto de vista hace que el dominio de *editor* no sea atómico.

La Figura 9.1 muestra una relación de ejemplo, *libros*.

Por simplificar se da por supuesto que el título del libro lo identifica de manera única¹. Se puede representar, entonces, la misma información usando el esquema siguiente:

- *autores(título, autor, posición)*
- *palabras_clave(título, palabra_clave)*
- *libros4(título, nombre_editor, sucursal_editor)*

El esquema anterior satisface la 4NF. La Figura 9.2 muestra la representación normalizada de los datos de la Figura 9.1.

Aunque la base de datos de libros de ejemplo puede expresarse de manera adecuada sin necesidad de usar relaciones anidadas, su uso lleva a un modelo más fácil de comprender. El usuario típico o el programador de sistemas de recuperación de la información piensa en la base de datos en términos de libros que tienen conjuntos de autores, como los modelos de diseño que no se hallan en la 1FN. El diseño de la 4FN exige consultas que reúnan varias relaciones, mientras que los diseños que no se hallan en la 1FN hacen más fáciles muchos tipos de consultas.

Por otro lado, en otras situaciones puede resultar más conveniente usar una representación en la primera forma normal en lugar de conjuntos. Por ejemplo, considérese la relación *impositor* del ejemplo bancario. La relación es de varios a varios entre *clientes* y *cuentas*. Teóricamente, se podría almacenar un conjunto de cuentas con cada cliente, o un conjunto de clientes con cada cuenta, o ambas cosas. Si se almacenaran ambas cosas, se tendría redundancia de los datos (la relación de un cliente concreto con una cuenta dada se almacenaría dos veces).

La posibilidad de usar tipos de datos complejos como los conjuntos y los arrays puede resultar útil en muchas aplicaciones, pero se debe usar con cuidado.

9.3 Tipos estructurados y herencia en SQL

Antes de SQL:1999 el sistema de tipos de SQL consistía en un conjunto bastante sencillo de tipos predefinidos. SQL:1999 añadió un sistema de tipos extenso a SQL, lo que permite los tipos estructurados y la herencia de tipos.

1. Esta suposición no se cumple en el mundo real. Los libros se suelen identificar por un número de ISBN de diez cifras que identifica de manera única cada libro publicado.

<i>título</i>	<i>array_autores</i>	<i>editor</i> (<i>nombre, sucursal</i>)	<i>conjunto_palabras_clave</i>
Compiladores	[Gómez, Santos]	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Redes	[Santos, Escudero]	(Oxford, Londres)	{Internet, Web}

Figura 9.1 Relación de libros que no está en la 1FN, *libros*.

título	autor	posición
Compiladores	Gómez	1
Compiladores	Santos	2
Redes	Santos	1
Redes	Escudero	2

autores

título	palabra_clave
Compiladores	análisis sintáctico
Compiladores	análisis
Redes	Internet
Redes	Web

palabras_clave

título	nombre_editor	sucursal_editor
Compiladores	McGraw-Hill	Nueva York
Redes	Oxford	Londres

libros4

Figura 9.2 Versión en la 4FN de la relación *libros*.

9.3.1 Tipos estructurados

Los tipos estructurados permiten representar directamente los atributos compuestos de los diagramas E-R. Por ejemplo, se puede definir el siguiente tipo estructurado para representar el atributo compuesto *nombre* con los atributos componentes *nombredepila* y *apellidos*:

```
create type Nombre as
  (nombredepila varchar(20),
  apellidos varchar(20))
final
```

De manera parecida, el tipo estructurado siguiente puede usarse para representar el atributo compuesto *dirección*:

```
create type Dirección as
  (calle varchar(20),
  ciudad varchar(20),
  códigopostal varchar(9))
not final
```

En SQL estos tipos se denominan **tipos definidos por el usuario**. La definición anterior corresponde al diagrama E-R de la Figura 6.4. Las especificaciones **final** y **not final** están relacionadas con la subtipificación, que se describirá más adelante, en el Apartado 9.3.22.² Ahora se pueden usar esos tipos para crear atributos compuestos en las relaciones, con sólo declarar que un atributo es de uno de estos tipos. Por ejemplo, se puede crear la tabla *cliente* de la manera siguiente:

```
create table cliente (
  nombre Nombre,
  dirección Dirección,
  fechaDeNacimiento date)
```

2. La especificación **final** de *Nombre* indica que no se pueden crear subtipos de *nombre*, mientras que la especificación **not final** de *Dirección* indica que se pueden crear subtipos de *dirección*.

Se puede tener acceso a los componentes de los atributos compuestos usando la notación “punto”; por ejemplo, *nombre.nombredelpila* devuelve el componente nombre de pila del atributo nombre. El acceso al atributo *nombre* devolvería un valor del tipo estructurado *Nombre*.

También se puede crear una tabla cuyas filas sean de un tipo definido por el usuario. Por ejemplo, se puede definir el tipo *TipoCliente* y crear la tabla *cliente* de la manera siguiente:

```
create type TipoCliente as (
    nombre Nombre,
    dirección Dirección,
    fechaDeNacimiento date)
not final
create table cliente of TipoCliente
```

Una manera alternativa de definir los atributos compuestos en SQL es usar **tipos de fila** sin nombre. Por ejemplo, la relación que representa la información del cliente se podría haber creado usando tipos de fila de la manera siguiente:

```
create table cliente (
    nombre row (nombredelpila varchar(20),
        apellidos varchar(20)),
    dirección row (calle varchar(20),
        ciudad varchar(20),
        códigoPostal varchar(9)),
    fechaDeNacimiento date)
```

Esta definición es equivalente a la anterior definición de la tabla, salvo en que los atributos *nombre* y *dirección* tienen tipos sin nombre y las filas de la tabla también tienen un tipo sin nombre.

La consulta siguiente ilustra la manera de tener acceso a los atributos componentes de los atributos compuestos. La consulta busca el apellido y la ciudad de cada cliente.

```
select nombre.apellidos, dirección.ciudad
from cliente
```

Sobre los tipos estructurados se pueden definir **métodos**. Los métodos se declaran como parte de la definición de los tipos de los tipos estructurados:

```
create type TipoCliente as (
    nombre Nombre,
    dirección Dirección,
    fechaDeNacimiento date)
not final
method edadAFecha(aFecha date)
    returns interval year
```

El cuerpo del método se crea por separado:

```
create instance method edadAFecha (aFecha date)
    returns interval year
    for TipoCliente
begin
    return aFecha - self.fechaDeNacimiento;
end
```

Téngase en cuenta que la cláusula **for** indica el tipo al que se aplica el método, mientras que la palabra clave **instance** indica que el método se ejecuta sobre un ejemplar del tipo *Cliente*. La variable **self** hace referencia al ejemplar de *Cliente* sobre el que se invoca el método. El cuerpo del método puede contener

instrucciones procedimentales, que ya se han visto en el Apartado 4.6. Los métodos pueden actualizar los atributos del ejemplar sobre el que se ejecutan.

Los métodos se pueden invocar sobre los ejemplares de los tipos. Si se hubiera creado la tabla *cliente* del tipo *TipoCliente*, se podría invocar el método *edadAFecha ()* como se ilustra a continuación, para averiguar la edad de cada cliente.

```
select nombre.apellidos, edadAFecha(current_date)
from cliente
```

En SQL:1999 se usan **funciones constructoras** para crear valores de los tipos estructurados. Las funciones con el mismo nombre que un tipo estructurado son funciones constructoras de ese tipo estructurado. Por ejemplo, se puede declarar una función constructora para el tipo *Nombre* de esta manera:

```
create function Nombre (nombredelpila varchar(20), apellidos varchar(20))
returns Nombre
begin
    set self.nombredelpila = nombredelpila;
    set self.apellidos = apellidos;
end
```

Se puede usar **new Nombre ('Martín', 'Gómez')** para crear un valor del tipo *Nombre*.

Se puede crear un valor de fila haciendo una relación de sus atributos entre paréntesis. Por ejemplo, si se declara el atributo *nombre* como tipo de fila con los componentes *nombredelpila* y *apellidos*, se puede crear para él este valor:

```
('Ted', 'Codd')
```

sin necesidad de función constructora.

De manera predeterminada, cada tipo estructurado tiene una función constructora sin argumentos, que configura los atributos con sus valores predeterminados. Cualquier otra función constructora hay que crearla de manera explícita. Puede haber más de una función constructora para el mismo tipo estructurado; aunque tengan el mismo nombre, deben poder distinguirse por el número y tipo de sus argumentos.

La instrucción siguiente ilustra la manera de crear una nueva tupla de la relación *Cliente*. Se da por supuesto que se ha definido una función constructora para *Dirección*, igual que la función constructora que se definió para *Nombre*.

```
insert into Cliente
values
  (new Nombre('Martín', 'Gómez'),
   new Dirección('Calle Mayor, 20', 'Madrid', '28045'),
   date '22-8-1960')
```

9.3.2 Herencia de tipos

Supóngase que se tiene la siguiente definición de tipo para las personas:

```
create type Persona
  (nombre varchar(20),
  dirección varchar(20))
```

Puede que se desee almacenar en la base de datos información adicional sobre las personas que son estudiantes y sobre las que son profesores. Dado que los estudiantes y los profesores también son personas, se puede usar la herencia para definir en SQL los tipos estudiante y profesor:

```

create type Estudiante
  under Persona
    (grado varchar(20),
     departamento varchar(20))
create type Profesor
  under Persona
    (sueldo integer,
     departamento varchar(20))

```

Tanto *Estudiante* como *Profesor* heredan los atributos de *Persona*—es decir, *nombre* y *dirección*. Se dice que *Estudiante* y *Profesor* son subtipos de *Persona*, y *Persona* es un supertipo de *Estudiante* y de *Profesor*.

Los métodos de los tipos estructurados se heredan por sus subtipos, igual que los atributos. Sin embargo, cada subtipo puede redefinir el efecto de los métodos volviendo a declararlos, usando **overriding method** en lugar de **method** en la declaración del método.

La norma de SQL también exige un campo adicional al final de la definición de los tipos, cuyo valor es **final** o **not final**. La palabra clave **final** indica que no se pueden crear subtipos a partir del tipo dado, mientras que **not final** indica que se pueden crear subtipos. Ahora, supóngase que se desea almacenar información sobre los profesores ayudantes, que son a la vez estudiantes y profesores, quizás incluso en departamentos diferentes. Esto se puede hacer si el sistema de tipos soporta **herencia múltiple**, por la que se pueden declarar los tipos como subtipos de varios tipos. Téngase en cuenta que la norma de SQL (hasta las versiones SQL:1999 y SQL:2003, al menos) no soporta la herencia múltiple, aunque puede que sí que la soporten versiones futuras.

Por ejemplo, si el sistema de tipos soporta la herencia múltiple, se puede definir el tipo para los profesores ayudantes de la manera siguiente:

```

create type ProfesorAyudante
  under Estudiante, Profesor

```

ProfesorAyudante heredará todos los atributos de *Estudiante* y de *Profesor*. No obstante, hay un problema, ya que los atributos *nombre*, *dirección* y *departamento* están presentes tanto en *Estudiante* como en *Profesor*.

Los atributos *nombre* y *dirección* se heredan realmente de una fuente común, *Persona*. Por tanto, no hay ningún conflicto causado por heredarlos de *Estudiante* y de *Profesor*. Sin embargo, el atributo *departamento* se define por separado en *Estudiante* y en *Profesor*. De hecho, cada profesor ayudante puede ser estudiante en un departamento y profesor en otro. Para evitar conflictos entre las dos apariciones de *departamento*, se pueden rebautizar usando una cláusula **as**, como en la definición del tipo *ProfesorAyudante*:

```

create type ProfesorAyudante
  under Estudiante with (departamento as departamento_estudiante),
  Profesor with (departamento as departamento_profesor)

```

Hay que tener en cuenta nuevamente que SQL sólo soporta la herencia simple—es decir, cada tipo sólo se puede heredar de un único tipo, la sintaxis usada es como la de los ejemplos anteriores. La herencia múltiple, como en el ejemplo de *ProfesorAyudante*, no está soportada en SQL.

En SQL, como en la mayor parte del resto de los lenguajes, el valor de un tipo estructurado debe tener exactamente un “tipo más concreto”. Es decir, cada valor debe asociarse, al crearlo, con un tipo concreto, denominado su **tipo más concreto**. Mediante la herencia también se asocia con cada uno de los supertipos de su tipo más concreto. Por ejemplo, supóngase que una entidad es tanto del tipo *Persona* como del tipo *Estudiante*. Entonces, el tipo más concreto de la entidad es *Estudiante*, ya que *Estudiante* es un subtipo de *Persona*. Sin embargo, una entidad no puede ser de los tipos *Estudiante* y *Profesor*, a menos que tenga un tipo, como *ProfesorAyudante*, que sea subtipo de *Profesor* y de *Estudiante* (lo cual no es posible en SQL, ya que SQL no soporta la herencia múltiple).

9.4 Herencia de tablas

Las subtablas de SQL se corresponden con el concepto de especialización/generalización de E-R. Por ejemplo, supóngase que se define la tabla *personas* de la manera siguiente:

```
create table personas of Persona
```

A continuación se pueden definir las tablas *estudiantes* y *profesores* como **subtablas** de *personas*, de la manera siguiente:

```
create table estudiantes of Estudiante
    under personas
create table profesores of Profesor
    under personas
```

Los tipos de las subtablas deben ser subtipos del tipo de la tabla madre. Por tanto, todos los atributos presentes en *personas* también están presentes en las subtablas.

Además, cuando se declaran *estudiantes* y *profesores* como subtablas de *personas*, todas las tuplas presentes en *estudiantes* o en *profesores* pasan a estar también presentes de manera implícita en *personas*. Por tanto, si una consulta usa la tabla *personas*, no sólo encuentra tuplas directamente insertadas en esa tabla, sino también tuplas insertadas en sus subtablas, es decir, *estudiantes* y *profesores*. No obstante, esa consulta sólo puede tener acceso a los atributos que están presentes en *personas*.

SQL permite hallar tuplas que se encuentran en *personas* pero no en sus subtablas usando en las consultas “**only** *personas*” en lugar de *personas*. La palabra clave **only** también puede usarse en las sentencias **delete** y **update**. Sin la palabra clave **only**, la instrucción **delete** aplicada a una supertabla, como *personas*, también borra las tuplas que se insertaron originalmente en las subtablas (como *estudiantes*); por ejemplo, la instrucción

```
delete from personas where P
```

borrará todas las tuplas de la tabla *personas*, así como de sus subtablas *estudiantes* y *profesores*, que satisfagan *P*. Si se añade la palabra clave **only** a la instrucción anterior, las tuplas que se insertaron en las subtablas no se ven afectadas, aunque satisfagan las condiciones de la cláusula **where**. Las consultas posteriores a la supertabla seguirán encontrando esas tuplas.

Teóricamente, la herencia múltiple es posible con las tablas, igual que con los tipos. Por ejemplo, se puede crear una tabla del tipo *ProfesorAyudante*:

```
create table profesores_ayudantes
of ProfesorAyudante
under estudiantes, profesores
```

Como consecuencia de la declaración, todas las tuplas presentes en la tabla *profesores_ayudantes* también se hallan presentes de manera implícita en las tablas *profesores* y *estudiantes* y, a su vez, en la tabla *personas*. Hay que tener en cuenta, no obstante, que SQL no soporta la herencia múltiple de tablas.

Existen varios requisitos de consistencia para las subtablas. Antes de definir las restricciones, es necesaria una definición: se dice que las tuplas de una subtabla se **corresponden** con las tuplas de la tabla madre si tienen el mismo valor para todos los atributos heredados. Por tanto, las tuplas correspondientes representan a la misma entidad.

Los requisitos de consistencia de las subtablas son:

1. Cada tupla de la supertabla puede corresponderse, como máximo, con una tupla de cada una de sus subtablas inmediatas.
2. SQL posee una restricción adicional que hace que todas las tuplas que se corresponden entre sí deben proceder de una tupla (insertada en una tabla).

Por ejemplo, sin la primera condición, se podrían tener dos tuplas de *estudiantes* (o de *profesores*) que correspondieran a la misma persona.

La segunda condición excluye que haya una tupla de *personas* correspondiente a una tupla de *estudiantes* y a una tupla de *profesores*, a menos que todas esas tuplas se hallen presentes de manera implícita porque se haya insertado una tupla en la tabla *profesores_ayudantes*, que es subtabla tanto de *profesores* como de *estudiantes*.

Dado que SQL no soporta la herencia múltiple, la segunda condición impide realmente que ninguna persona sea a la vez profesor y estudiante. Aunque se soportara la herencia múltiple, surgiría el mismo problema si estuviera ausente la subtabla *profesores_ayudantes*. Evidentemente, resultaría útil modelar una situación en la que alguien pudiera ser a la vez profesor y estudiante, aunque no se hallara presente la subtabla *profesores_ayudantes*. Por tanto, puede resultar útil eliminar la segunda restricción de consistencia. Hacerlo permitiría que cada objeto tuviera varios tipos, sin necesidad de que tuviera un tipo más concreto.

Por ejemplo, supóngase que se vuelve a tener el tipo *Persona*, con los subtipos *Estudiante* y *Profesor*, y la tabla correspondiente *personas*, con las subtablas *profesores* y *estudiantes*. Se puede tener una tupla de *profesores* y una tupla de *estudiantes* correspondientes a la misma tupla de *personas*. No hace falta tener el tipo *ProfesorAyudante*, que es subtipo de *Estudiante* y de *Profesor*. No hace falta crear el tipo *ProfesorAyudante* a menos que se desee almacenar atributos adicionales o redefinir los métodos de manera específica para las personas que sean a la vez estudiantes y profesores.

No obstante, hay que tener en cuenta que, por desgracia, SQL prohíbe las situaciones de este tipo debido al segundo requisito de consistencia. Ya que SQL tampoco soporta la herencia múltiple, no se puede usar la herencia para modelar una situación en la que alguien pueda ser a la vez estudiante y profesor. En consecuencia, las subtablas de SQL no se pueden usar para representar las especializaciones que se solapen de los modelos E-R.

Por supuesto, se pueden crear tablas diferentes para representar las especializaciones o generalizaciones que se solapan sin usar la herencia. El proceso ya se ha descrito anteriormente, en el Apartado 6.9.5. En el ejemplo anterior, se crearían las tablas *personas*, *estudiantes* y *profesores*, de las que las tablas *estudiantes* y *profesores* contendrían el atributo de clave primaria de *Persona* y otros atributos específicos de *Estudiante* y de *Profesor*, respectivamente. La tabla *personas* contendría la información sobre todas las personas, incluidos los estudiantes y los profesores. Luego habría que añadir las restricciones de integridad referencial correspondientes para garantizar que los estudiantes y los profesores también se hallen representados en la tabla *personas*.

En otras palabras, se puede crear una implementación mejorada del mecanismo de las subtablas mediante las características de SQL ya existentes, con algún esfuerzo adicional para la definición de la tabla, así como en el momento de las consultas para especificar las reuniones para el acceso a los atributos necesarios.

Para finalizar este apartado, hay que tener en cuenta que SQL define un nuevo privilegio denominado **under**, el cual es necesario para crear subtipos o subtablas bajo otro tipo o tabla. La razón de ser de este privilegio es parecida a la del privilegio **references**.

9.5 Tipos array y multiconjunto en SQL

SQL soporta dos tipos de conjuntos: arrays y multiconjuntos; los tipos array se añadieron en SQL:1999, mientras que los tipos multiconjunto se agregaron en SQL:2003. Recuérdese que un *multiconjunto* es un conjunto no ordenado, en el que cada elemento puede aparecer varias veces. Los multiconjuntos son como los conjuntos, salvo que los conjuntos permiten que cada elemento aparezca, como mucho, una vez.

Supóngase que se desea registrar información sobre libros, incluido un conjunto de palabras clave para cada libro. Supóngase también que se deseara almacenar el nombre de los autores de un libro en forma de array; a diferencia de los elementos de los multiconjuntos, los elementos de los arrays están ordenados, de modo que se puede distinguir el primer autor del segundo, etc. El ejemplo siguiente ilustra la manera en que se pueden definir en SQL estos atributos valorados como arrays y como multiconjuntos.

```

create type Editor as
  (nombre varchar(20),
   sucursal varchar(20))
create type Libro as
  (título varchar(20),
   array_autores varchar(20) array [10],
   fecha_publicación date,
   editor Editor,
   conjunto_palabras_clave varchar(20) multiset)
create table libros of Libro

```

La primera instrucción define el tipo denominado *Editor*, que tiene dos componentes: nombre y sucursal. La segunda instrucción define el tipo estructurado *Libro*, que contiene un *título*, un *array_autores*, que es un array de hasta diez nombres de autor, una fecha de publicación, un editor (del tipo *Editor*), y un multiconjunto de palabras clave. Finalmente, se crea la tabla *libros*, que contiene las tuplas del tipo *Libro*.

Téngase en cuenta que se ha usado un array, en lugar de un multiconjunto, para almacenar el nombre de los autores, ya que el orden de los autores suele tener cierta importancia, mientras que se considera que el orden de las palabras asociadas con el libro no es significativo.

En general, los atributos multivalorados de los esquemas E-R se pueden asignar en SQL a atributos valorados como multiconjuntos; si el orden es importante, se pueden usar los arrays de SQL en lugar de los multiconjuntos.

9.5.1 Creación y acceso a los valores de los conjuntos

En SQL:1999 se puede crear un array de valores de esta manera:

```
array[‘Silberschatz’, ‘Korth’, ‘Sudarshan’]
```

De manera parecida, se puede crear un multiconjunto de palabras clave de la manera siguiente:

```
multiset[‘computadora’, ‘base de datos’, ‘SQL’]
```

Por tanto, se puede crear una tupla del tipo definido por la relación *libros* como:

```
(‘Compiladores’, array[‘Gómez’, ‘Santos’], new Editor(‘McGraw-Hill’, ‘Nueva York’),
 multiset[‘análisis sintáctico’, ‘análisis’])
```

En este ejemplo se ha creado un valor para el atributo *Editor* mediante la invocación a una función *constructora* para *Editor* con los argumentos correspondientes. Téngase en cuenta que esta constructora para *Editor* se debe crear de manera explícita y no se halla presente de manera predeterminada; se puede declarar como la constructora para *Nombre*, que ya se ha visto en el Apartado 9.3.

Si se desea insertar la tupla anterior en la relación *libros*, se puede ejecutar la instrucción:

```

insert into libros
values
(‘Compiladores’, array[‘Gómez’, ‘Santos’],
 new Editor(‘McGraw-Hill’, ‘Nueva York’),
 multiset[‘análisis sintáctico’, ‘análisis’])

```

Se puede tener acceso a los elementos del array o actualizarlos especificando el índice del array, por ejemplo, *array_autores* [1].

9.5.2 Consulta de los atributos valorados como conjuntos

Ahora se considerará la forma de manejar los atributos que se valoran como conjuntos. Las expresiones que se valoran como conjuntos pueden aparecer en cualquier parte en la que pueda aparecer el nombre

de una relación, como las cláusulas **from**, como ilustran los siguientes párrafos. Se usará la tabla *libros* que ya se había definido anteriormente.

Si se desea averiguar todos los libros que tienen las palabras “base de datos” entre sus palabras clave se puede usar la consulta siguiente:

```
select título
from libros
where 'base de datos' in (unnest(conjunto_palabras_clave))
```

Obsérvese que se ha usado **unnest**(*conjunto_palabras_clave*) en una posición en la que SQL sin las relaciones anidadas habría exigido una subexpresión **select-from-where**.

Si se sabe que un libro concreto tiene tres autores, se puede escribir:

```
select array_autores[1], array_autores[2], array_autores[3]
from libros
where título = 'Fundamentos de bases de datos'
```

Ahora supóngase que se desea una relación que contenga parejas de la forma “título, nombre_autor” para cada libro y para cada uno de sus autores. Se puede usar esta consulta:

```
select L.título, A.autores
from libros as L, unnest(L.array_autores) as A(autores)
```

Dado que el atributo *array_autores* de *libros* es un campo que se valora como conjunto, **unnest**(*L.array_autores*) puede usarse en una cláusula **from** en la que se espere una relación. Téngase en cuenta que la variable tupla *B* es visible para esta expresión, ya que se ha definido *anteriormente* en la cláusula **from**.

Al desanidar un array la consulta anterior pierde información sobre el orden de los elementos del array. Se puede usar la cláusula **unnest with ordinality** para obtener esta información, como ilustra la consulta siguiente. Esta consulta se puede usar para generar la relación *autores*, que se ha visto anteriormente, a partir de la relación *libros*.

```
select título, A.autores, A.posición
from libros as L,
unnest(L.array_autores) with ordinality as A(autores, posición)
```

La cláusula **with ordinality** genera un atributo adicional que registra la posición del elemento en el array. Se puede usar una consulta parecida, pero sin la cláusula **with ordinality**, para generar la relación *palabras_clave*.

9.5.3 Anidamiento y desanidamiento

La transformación de una relación anidada en una forma con menos atributos de tipo relación (o sin ellos) se denomina **desanidamiento**. La relación *libros* tiene dos atributos, *array_autores* y *conjunto_palabras_clave*, que son conjuntos, y otros dos, *título* y *editor*, que no lo son. Supóngase que se desea convertir la relación en una sola relación plana, sin relaciones anidadas ni tipos estructurados como atributos. Se puede usar la siguiente consulta para llevar a cabo la tarea:

```
select título, A.autor, editor.nombre as nombre_editor, editor.sucursal
      as sucursal_editor, P.palabras_clave
from libros as L, unnest(L.array_autores) as A(autores),
      unnest(L.conjunto_palabras_clave) as P(palabras_clave)
```

La variable *L* de la cláusula **from** se declara para que tome valores de *libros*. La variable *A* se declara para que tome valores de los autores de *array_autores* para el libro *L* y *P* se declara para que tome valores de las palabras clave del *conjunto_palabras_clave* del libro *L*. La Figura 9.1 muestra un ejemplar de la relación *libros* y la Figura 9.3 muestra la relación, denominada *libros_plana*, que es resultado de la consulta

título	autor	nombre_editor	sucursal_editor	conjunto_palabras_clave
Compiladores	Gómez	McGraw-Hill	Nueva York	análisis sintáctico
Compiladores	Santos	McGraw-Hill	Nueva York	análisis sintáctico
Compiladores	Gómez	McGraw-Hill	Nueva York	análisis
Compiladores	Santos	McGraw-Hill	Nueva York	análisis
Redes	Santos	Oxford	Londres	Internet
Redes	Escudero	Oxford	Londres	Internet
Redes	Santos	Oxford	Londres	Web
Redes	Escudero	Oxford	Londres	Web

Figura 9.3 *libros_plana*: resultado del desanidamiento de los atributos *array_autores* y *conjunto_palabras_clave* de la relación *libros*.

anterior. Téngase en cuenta que la relación *libros_plana* se halla en 1FN, ya que todos sus atributos son atómicos.

El proceso inverso de transformar una relación en la 1FN en una relación anidada se denomina **anidamiento**. El anidamiento puede realizarse mediante una extensión de la agrupación en SQL. En el uso normal de la agrupación en SQL se crea (lógicamente) una relación multiconjunto temporal para cada grupo y se aplica una función de agregación a esa relación temporal para obtener un valor único (atómico). La función **collect** devuelve el multiconjunto de valores; en lugar de crear un solo valor se puede crear una relación anidada. Supóngase que se tiene la relación en 1FN *libros_plana*, tal y como se muestra en la Figura 9.3. La consulta siguiente anida la relación en el atributo *palabras_clave*:

```
select título, autores, Editor(nombre_editor, sucursal_editor) as editor,
       collect(palabras_clave) as conjunto_palabras_clave
  from libros_plana
 group by título, autor, editor
```

El resultado de la consulta a la relación *libros_plana* de la Figura 9.3 aparece en la Figura 9.4.

Si se desea anidar también el atributo *autores* en un multiconjunto, se puede usar la consulta:

```
select título, collect(autores) as conjunto_autores,
       Editor(nombre_editor, sucursal_editor) as editor,
       collect(palabra_clave) as conjunto_palabras_clave
  from libros_plana
 group by título, editor
```

Otro enfoque de la creación de relaciones anidadas es usar subconsultas en la cláusula **select**. Una ventaja del enfoque de las subconsultas es que se puede usar de manera opcional en la subconsulta una cláusula **order by** para generar los resultados en el orden deseado, lo que puede aprovecharse para crear un array. La siguiente consulta ilustra este enfoque; las palabras clave **array** y **multiset** especifican que se van a crear un array y un multiconjunto (respectivamente) a partir del resultado de las subconsultas.

título	autor	editor	conjunto_palabras_clave
		(nombre_editor, sucursal_editor)	
Compiladores	Gómez	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Compiladores	Santos	(McGraw-Hill, Nueva York)	{análisis sintáctico, análisis}
Redes	Santos	(Oxford, Londres)	{Internet, Web}
Redes	Escudero	(Oxford, Londres)	{Internet, Web}

Figura 9.4 Una versión parcialmente anidada de la relación *libros_plana*.

```

select título,
array( select autores
      from autores as A
      where A.título = L.título
      order by A.posición) as array_autores,
Editor(nombre_editor, sucursal_editor) as editor,
multiset( select palabra_clave
          from palabras_clave as P
          where P.título = L.título) as conjunto_palabras_clave,
from libros4 as L

```

El sistema ejecuta las subconsultas anidadas de la cláusula **select** para cada tupla generada por las cláusulas **from** and **where** de la consulta externa. Obsérvese que el atributo *L.título* de la consulta externa se usa en las consultas anidadas para garantizar que sólo se generen los conjuntos correctos de autores y de palabras clave para cada título.

SQL:2003 ofrece gran variedad de operadores para multiconjuntos, incluida la función **set(*M*)**, que calcula una versión libre duplicada del multiconjunto *M*, la operación agregada **intersection**, cuyo resultado es la intersección de todos los multiconjuntos de un grupo, la operación agregada **fusion**, que devuelve la unión de todos los multiconjuntos de un grupo, y el predicado **submultiset**, que comprueba si el multiconjunto está contenido en otro multiconjunto.

La norma de SQL no proporciona ningún medio para actualizar los atributos de los multiconjuntos, salvo asignarles valores nuevos. Por ejemplo, para borrar el valor *v* del atributo de multiconjunto *A* hay que definirlo como (*A except all multiset[v]*).

9.6 Identidad de los objetos y tipos de referencia en SQL

Los lenguajes orientados a objetos ofrecen la posibilidad de hacer referencia a objetos. Los atributos de un tipo dado pueden servir de referencia para los objetos de un tipo concreto. Por ejemplo, en SQL se puede definir el tipo *Departamento* con el campo *nombre* y el campo *director*, que es una referencia al tipo *Persona*, y la tabla *departamentos* del tipo *Departamento*, de la manera siguiente:

```

create type Departamento (
    nombre varchar(20),
    director ref(Persona) scope personas
)
create table departamentos of Departamento

```

En este caso, la referencia está restringida a las tuplas de la tabla *personas*. La restricción del **ámbito** (scope) de referencia a las tuplas de una tabla es obligatoria en SQL, y hace que las referencias se comporten como las claves externas.

Se puede omitir la declaración **scope personas** de la declaración de tipos y hacer, en su lugar, un añadido a la instrucción **create table**:

```

create table departamentos of Departamento
(director with options scope personas)

```

La tabla a la que se hace referencia debe tener un atributo que guarde el identificador de cada tupla. Ese atributo, denominado **atributo autorreferencial** (self-referential attribute), se declara añadiendo una cláusula **ref is** a la instrucción **create table**:

```

create table personas of Persona
ref is id_personal system generated

```

En este caso, *id_personal* es el nombre de un atributo, no una palabra clave, y la instrucción **create table** especifica que la base de datos genera de manera automática el identificador.

Para inicializar el atributo de referencia hay que obtener el identificador de la tupla a la que se va a hacer referencia. Se puede conseguir el valor del identificador de la tupla mediante una consulta. Por tanto, para crear una tupla con el valor de referencia, primero se puede crear la tupla con una referencia nula y luego definir la referencia de manera independiente:

```
insert into departamentos
    values ('CS', null)
update departamentos
    set director = (select p.id_personal
        from personal as p
        where nombre = 'Martín')
where nombre = 'CS'
```

Una alternativa a los identificadores generados por el sistema es permitir que los usuarios generen los identificadores. El tipo del atributo autorreferencial debe especificarse como parte de la definición de tipos de la tabla a la que se hace referencia, y la definición de la tabla debe especificar que la referencia está **generada por el usuario (user generated)**:

```
create type Persona
    (nombre varchar(20),
     dirección varchar(20))
ref using varchar(20)
create table personas of Persona
    ref is id_personal user generated
```

Al insertar tuplas en *personas* hay que proporcionar el valor del identificador:

```
insert into personas (id_personal, nombre, dirección) values
    ('01284567', 'Martín', 'Avenida del Segura, 23')
```

Ninguna otra tupla de *personas*, de sus supertablas ni de sus subtablas puede tener el mismo identificador. Por tanto, se puede usar el valor del identificador al insertar tuplas en *departamentos*, sin necesidad de más consultas para recuperarlo:

```
insert into departamentos
    values ('CS', '01284567')
```

Incluso es posible usar el valor de una clave primaria ya existente como identificador, incluyendo la cláusula **ref from** en la definición de tipos:

```
create type Persona
    (nombre varchar(20) primary key,
     dirección varchar(20))
ref from(nombre)
create table personas of Persona
    ref is id_personal derived
```

Téngase en cuenta que la definición de la tabla debe especificar que las referencia es derivada, y debe seguir especificando el nombre de un atributo autorreferencial. Al insertar una tupla de *departamentos* se puede usar

```
insert into departamentos
    values ('CS', 'Martín')
```

En SQL:1999 las referencias se desvinculan mediante el símbolo \rightarrow . Considérese la tabla *departamentos* que se ha definido anteriormente. Se puede usar esta consulta para averiguar el nombre y la dirección de los directores de todos los departamentos:

```
select director->nombre, director->dirección
from departamentos
```

Las expresiones como “*director->nombre*” se denominan **expresiones de camino**.

Dado que *director* es una referencia a una tupla de la tabla *personas*, el atributo *nombre* de la consulta anterior es el atributo *nombre* de la tupla de la tabla *personas*. Se pueden usar las referencias para ocultar las operaciones de reunión; en el ejemplo anterior, sin las referencias, el campo *director* de *departamento* se declararía como clave externa de la tabla *personas*. Para averiguar el nombre y la dirección del director de un departamento, hace falta una reunión explícita de las relaciones *departamentos* y *personas*. El uso de referencias simplifica considerablemente la consulta.

Se puede usar la operación **deref** para devolver la tupla a la que señala una referencia y luego tener acceso a sus atributos, como se muestra a continuación.

```
select deref(director).nombre
from departamentos
```

9.7 Implementación de las características O-R

Los sistemas de bases de datos orientadas a objetos son básicamente extensiones de los sistemas de bases de datos relacionales ya existentes. Las modificaciones resultan claramente necesarias en muchos niveles del sistema de bases de datos. Sin embargo, para minimizar las modificaciones en el código del sistema de almacenamiento (almacenamiento de relaciones, índices, etc.), los tipos de datos complejos soportados por los sistemas orientados a objetos se pueden traducir al sistema de tipos más sencillo de las bases de datos relacionales.

Para comprender la manera de hacer esta traducción, sólo hace falta mirar al modo en que algunas características del modelo E-R se traducen en relaciones. Por ejemplo, los atributos multivalorados del modelo E-R se corresponden con los atributos valorados como multiconjuntos del modelo relacional orientado a objetos. Los atributos compuestos se corresponden grosso modo con los tipos estructurados. Las jerarquías ES del modelo E-R se corresponden con la herencia de tablas del modelo relacional orientado a objetos.

Las técnicas para convertir las características del modelo E-R en tablas, que se estudiaron en el Apartado 6.9, se pueden usar, con algunas extensiones, para traducir los datos orientados a objetos a datos orientados a relaciones en el nivel de almacenamiento.

Las subtablas se pueden almacenar de manera eficiente, sin réplica de todos los campos heredados, de una de estas maneras:

- Cada tabla almacena la clave primaria (que puede haber heredado de una tabla madre) y los atributos que se definen de localmente. No hace falta almacenar los atributos heredados (que no sean la clave primaria), se pueden obtener mediante una reunión con la supertabla, de acuerdo con la clave primaria.
- Cada tabla almacena todos los atributos heredados y definidos localmente. Cuando se inserta una tupla, sólo se almacena en la tabla en la que se inserta, y su presencia se infiere en cada una de las supertablas. El acceso a todos los atributos de las tuplas es más rápido, ya que no hace falta ninguna reunión.

No obstante, en caso de que el sistema de tipos permita que cada entidad se represente en dos subtablas sin estar presente en una subtabla común a ambas, esta representación puede dar lugar a la réplica de información. Además, resulta difícil traducir las claves externas que hacen referencia a una supertabla en restricciones de las subtablas; para implementar de manera eficiente esas claves externas hay que definir la supertabla como vista, y el sistema de bases de datos tiene que soportar las claves externas en las vistas.

Las implementaciones pueden decidir representar los tipos arrays y multiconjuntos directamente o usar internamente una representación normalizada. Las representaciones normalizadas tienden a ocupar más espacio y exigen un coste adicional en reuniones o agrupamientos para reunir los datos en

arrays o en multiconjuntos. Sin embargo, puede que las representaciones normalizadas resulten más sencillas de implementar.

Las interfaces de programas de aplicación ODBC y JDBC se han extendido para recuperar y almacenar tipos estructurados; por ejemplo, JDBC ofrece el método `getObject()`, que es parecido a `getString()` pero devuelve un objeto Java `Struct`, a partir del cual se pueden extraer los componentes del tipo estructurado. También es posible asociar clases de Java con tipos estructurados de SQL, y JDBC puede realizar la conversión entre los tipos. Véase el manual de referencia de ODBC o de JDBC para obtener más detalles.

9.8 Lenguajes de programación persistentes

Los lenguajes de las bases de datos se diferencian de los lenguajes de programación tradicionales en que trabajan directamente con datos que son persistentes; es decir, los datos siguen existiendo una vez que el programa que los creó haya concluido. Las relaciones de las bases de datos y las tuplas de las relaciones son ejemplos de datos persistentes. Por el contrario, los únicos datos persistentes con los que los lenguajes de programación tradicionales trabajan directamente son los archivos.

El acceso a las bases de datos es sólo un componente de las aplicaciones del mundo real. Mientras que los lenguajes para el tratamiento de datos como SQL son bastante efectivos en el acceso a los datos, se necesita un lenguaje de programación para implementar otros componentes de las aplicaciones como las interfaces de usuario o la comunicación con otras computadoras. La manera tradicional de realizar las interfaces de las bases de datos con los lenguajes de programación es incorporar SQL dentro del lenguaje de programación.

Los **lenguajes de programación persistentes** son lenguajes de programación extendidos con estructuras para el tratamiento de los datos persistentes. Los lenguajes de programación persistentes pueden distinguirse de los lenguajes con SQL incorporado, al menos, de dos maneras:

1. En los lenguajes incorporados el sistema de tipos del lenguaje anfitrión suele ser diferente del sistema de tipos del lenguaje para el tratamiento de los datos. Los programadores son responsables de las conversiones de tipos entre el lenguaje anfitrión y SQL. Hacer que los programadores lleven a cabo esta tarea presenta varios inconvenientes:
 - El código para la conversión entre objetos y tuplas opera fuera del sistema de tipos orientado a objetos y, por tanto, tiene más posibilidades de presentar errores no detectados.
 - La conversión en la base de datos entre el formato orientado a objetos y el formato relacional de las tuplas necesita gran cantidad de código. El código para la conversión de formatos, junto con el código para cargar y descargar los datos de la base de datos, puede suponer un porcentaje significativo del código total necesario para la aplicación.

Por el contrario, en los lenguajes de programación persistentes, el lenguaje de consultas se halla totalmente integrado con el lenguaje anfitrión y ambos comparten el mismo sistema de tipos. Los objetos se pueden crear y guardar en la base de datos sin ninguna modificación explícita del tipo o del formato; los cambios de formato necesarios se realizan de manera transparente.

2. Los programadores que usan lenguajes de consultas incorporados son responsables de la escritura de código explícito para la búsqueda en la memoria de los datos de la base de datos. Si se realizan actualizaciones, los programadores deben escribir explícitamente código para volver a guardar los datos actualizados en la base de datos.

Por el contrario, en los lenguajes de programación persistentes, los programadores pueden trabajar con datos persistentes sin tener que escribir explícitamente código para buscarlos en la memoria o volver a guardarlos en el disco.

En este apartado se describe la manera en que se pueden extender los lenguajes de programación orientados a objetos, como C++ y Java, para hacerlos lenguajes de programación persistentes. Las características de estos lenguajes permiten que los programadores trabajen con los datos directamente desde el lenguaje de programación, sin tener que recurrir a lenguajes de tratamiento de datos como SQL. Por tanto, ofrecen una integración más estrecha de los lenguajes de programación con las bases de datos que, por ejemplo, SQL incorporado.

Sin embargo, los lenguajes de programación persistentes presentan ciertos inconvenientes que hay que tener presentes al decidir si conviene usarlos. Dado que los lenguajes de programación suelen ser potentes, resulta relativamente sencillo cometer errores de programación que dañen las bases de datos. La complejidad de los lenguajes hace que la optimización automática de alto nivel, como la reducción de E/S de disco, resulte más difícil. En muchas aplicaciones el soporte de las consultas declarativas resulta de gran importancia, pero los lenguajes de programación persistentes no soportan bien actualmente las consultas declarativas.

En este capítulo se describen varios problemas teóricos que hay que abordar a la hora de añadir la persistencia a los lenguajes de programación ya existentes. En primer lugar se abordan los problemas independientes de los lenguajes y, en apartados posteriores, se tratan problemas que son específicos de los lenguajes C++ y Java. No obstante, no se tratan los detalles de las extensiones de los lenguajes; aunque se han propuesto varias normas, ninguna ha tenido una aceptación universal. Véanse las referencias de las notas bibliográficas para saber más sobre extensiones de lenguajes concretas y más detalles de sus implementaciones.

9.8.1 Persistencia de los objetos

Los lenguajes de programación orientados a objetos ya poseen un concepto de objeto, un sistema de tipos para definir los tipos de los objetos y constructores para crearlos. Sin embargo, esos objetos son *transitorios*; desaparecen en cuanto finaliza el programa, igual que ocurre con las variables de los programas en Pascal o en C. Si se desea transformar uno de estos lenguajes en un lenguaje para la programación de bases de datos, el primer paso consiste en proporcionar una manera de hacer persistentes a los objetos. Se han propuesto varios enfoques.

- **Persistencia por clases.** El enfoque más sencillo, pero el menos conveniente, consiste en declarar que una clase es persistente. Todos los objetos de la clase son, por tanto, persistentes de manera predeterminada. Todos los objetos de las clases no persistentes son transitorios.

Este enfoque no es flexible, dado que suele resultar útil disponer en una misma clase tanto de objetos transitorios como de objetos persistentes. Muchos sistemas de bases de datos orientados a objetos interpretan la declaración de que una clase es persistente como si se afirmara que los objetos de la clase pueden hacerse persistentes, en vez de que todos los objetos de la clase son persistentes. Estas clases se pueden denominar con más propiedad clases “que pueden ser persistentes”.

- **Persistencia por creación.** En este enfoque se introduce una sintaxis nueva para crear los objetos persistentes mediante la extensión de la sintaxis para la creación de los objetos transitorios. Por tanto, los objetos son persistentes o transitorios en función de la forma de crearlos. Varios sistemas de bases de datos orientados a objetos siguen este enfoque.
- **Persistencia por marcas.** Una variante del enfoque anterior es marcar los objetos como persistentes después de haberlos creado. Todos los objetos se crean como transitorios pero, si un objeto tiene que persistir más allá de la ejecución del programa, hay que marcarlo como persistente de manera explícita antes de que éste concluya. Este enfoque, a diferencia del anterior, pospone la decisión sobre la persistencia o la transitoriedad hasta después de la creación del objeto.
- **Persistencia por alcance.** Uno o varios objetos se declaran objetos persistentes (objetos raíz) de manera explícita. Todos los demás objetos serán persistentes si (y sólo si) se pueden alcanzar desde algún objeto raíz mediante una secuencia de una o varias referencias.

Por tanto, todos los objetos a los que se haga referencia desde (es decir, cuyos identificadores de objetos se guarden en) los objetos persistentes raíz serán persistentes. Pero también lo serán todos los objetos a los que se haga referencia desde ellos, y los objetos a los que éstos últimos hagan referencia serán también persistentes, etc.

Una ventaja de este esquema es que resulta sencillo hacer que sean persistentes estructuras de datos completas con sólo declarar como persistente su raíz. Sin embargo, el sistema de bases de datos sufre la carga de tener que seguir las cadenas de referencias para detectar los objetos que son persistentes, y eso puede resultar costoso.

9.8.2 Identidad de los objetos y punteros

En los lenguajes de programación orientados a objetos que no se han extendido para tratar la persistencia, cuando se crea un objeto el sistema devuelve un identificador del objeto transitorio. Los identificadores de objetos transitorios sólo son válidos mientras se ejecuta el programa que los ha creado; después de que concluya ese programa, el objeto se borra y el identificador pierde su sentido. Cuando se crea un objeto persistente se le asigna un identificador de objeto persistente.

El concepto de identidad de los objetos tiene una relación interesante con los punteros de los lenguajes de programación. Una manera sencilla de conseguir una identidad intrínseca es usar los punteros a las ubicaciones físicas de almacenamiento. En concreto, en muchos lenguajes orientados a objetos como C++, los identificadores de los objetos son en realidad punteros internos de la memoria.

Sin embargo, la asociación de los objetos con ubicaciones físicas de almacenamiento puede variar con el tiempo. Hay varios grados de permanencia de las identidades:

- **Dentro de los procedimientos.** La identidad sólo persiste durante la ejecución de un único procedimiento. Un ejemplo de identidad dentro de los programas son las variables locales de los procedimientos.
- **Dentro de los programas.** La identidad sólo persiste durante la ejecución de un único programa o de una única consulta. Un ejemplo de identidad dentro de los programas son las variables globales de los lenguajes de programación. Los punteros de la memoria principal o de la memoria virtual sólo ofrecen identidad dentro de los programas.
- **Entre programas.** La identidad persiste de una ejecución del programa a otra. Los punteros a los datos del sistema de archivos del disco ofrecen identidad entre los programas, pero pueden cambiar si se modifica la manera en que los datos se guardan en el sistema de archivos.
- **Persistente.** La identidad no sólo persiste entre las ejecuciones del programa sino también entre reorganizaciones estructurales de los datos. Es la forma persistente de identidad necesaria para los sistemas orientados a objetos.

En las extensiones persistentes de los lenguajes como C++, los identificadores de objetos de los objetos persistentes se implementan como “punteros persistentes”. Un *puntero persistente* es un tipo de puntero que, a diferencia de los punteros internos de la memoria, sigue siendo válido después del final de la ejecución del programa y después de algunas modalidades de reorganización de los datos. Los programadores pueden usar los punteros persistentes del mismo modo que usan los punteros internos de la memoria en los lenguajes de programación. Conceptualmente los punteros persistentes se pueden considerar como punteros a objetos de la base de datos.

9.8.3 Almacenamiento y acceso a los objetos persistentes

¿Qué significa guardar un objeto en una base de datos? Evidentemente, hay que guardar por separado la parte de datos de cada objeto. Lógicamente, el código que implementa los métodos de las clases debe guardarse en la base de datos como parte de su esquema, junto con las definiciones de tipos de las clases. Sin embargo, muchas implementaciones se limitan a guardar el código en archivos externos a la base de datos para evitar tener que integrar el software del sistema, como los compiladores, con el sistema de bases de datos.

Hay varias maneras de hallar los objetos de la base de datos. Una manera es dar nombres a los objetos, igual que se hace con los archivos. Este enfoque funciona con un número de objetos relativamente pequeño, pero no resulta práctico para millones de objetos. Una segunda manera es exponer los identificadores de los objetos o los punteros persistentes a los objetos, que pueden guardarse en el exterior. A diferencia de los nombres, los punteros no tienen por qué ser fáciles de recordar y pueden ser, incluso, punteros físicos internos de la base de datos.

Una tercera manera es guardar conjuntos de objetos y permitir que los programas iteren sobre ellos para buscar los objetos deseados. Los conjuntos de objetos pueden a su vez modelarse como objetos de un *tipo conjunto*. Entre los tipos de conjuntos están los conjuntos, los multiconjuntos (es decir, conjuntos con varias apariciones posibles de un mismo valor), las listas, etc. Un caso especial de conjunto son

las *extensiones de clases*, que son el conjunto de todos los objetos pertenecientes a una clase. Si hay una extensión de clase para una clase dada, siempre que se crea un objeto de la clase ese objeto se inserta en la extensión de clase de manera automática; y, siempre que se borra un objeto, éste se elimina de la extensión de clase. Las extensiones de clases permiten que las clases se traten como relaciones en el sentido de que es posible examinar todos los objetos de una clase, igual que se pueden examinar todas las tuplas de una relación.

La mayor parte de los sistemas de bases de datos orientados a objetos soportan las tres maneras de acceso a los objetos persistentes. Dan identificadores a todos los objetos. Generalmente sólo dan nombre a las extensiones de las clases y a otros objetos de tipo conjunto y, quizás, a otros objetos seleccionados, pero no a la mayor parte de los objetos. Las extensiones de las clases suelen conservarse para todas las clases que puedan tener objetos persistentes pero, en muchas de las implementaciones, las extensiones de las clases sólo contienen los objetos persistentes de cada clase.

9.8.4 Sistemas persistentes de C++

En los últimos años han aparecido varias bases de datos orientadas a objetos basadas en las extensiones persistentes de C++ (véanse las notas bibliográficas). Hay diferencias entre ellas en términos de la arquitectura de los sistemas pero tienen muchas características comunes en términos del lenguaje de programación.

Varias de las características orientadas a objetos del lenguaje C++ ayudan a proporcionar un buen soporte para la persistencia sin modificar el propio lenguaje. Por ejemplo, se puede declarar una clase denominada **Persistent_Object** (objeto persistente) con los atributos y los métodos para dar soporte la persistencia; cualquier otra clase que deba ser persistente puede hacerse subclase de esta clase y heredará, por tanto, el soporte de la persistencia. El lenguaje C++ (igual que otros lenguajes modernos de programación) permite también redefinir los nombres de las funciones y los operadores estándar—como +, -, el operador de desvinculación de los punteros ->, etc.—en función del tipo de operandos a los que se aplican. Esta posibilidad se denomina *sobrecarga*; se usa para redefinir los operadores para que se comporten de la manera deseada cuando operan con objetos persistentes.

Proporcionar apoyo a la persistencia mediante las bibliotecas de clases presenta la ventaja de que sólo se realizan cambios mínimos en C++; además, resulta relativamente fácil de implementar. Sin embargo, presenta el inconveniente de que los programadores tienen que usar mucho más tiempo para escribir los programas que trabajan con objetos persistentes y de que no les resulta sencillo especificar las restricciones de integridad del esquema ni ofrecer soporte para las consultas declarativas. Algunas implementaciones persistentes de C++ soportan extensiones de la sintaxis de C++ para facilitar estas tareas.

Es necesario abordar los siguientes problemas a la hora de añadir soporte a la persistencia a C++ (y a otros lenguajes):

- **Punteros persistentes.** Se debe definir un nuevo tipo de datos para que represente los punteros persistentes. Por ejemplo, la norma ODMG de C++ define la clase de plantillas `d_Ref< T >` para que represente los punteros persistentes a la clase `T`. El operador de desvinculación para esta clase se vuelve a definir para que capture el objeto del disco (si no se halla ya presente en la memoria) y devuelva un puntero de la memoria al búfer en el que se ha capturado el objeto. Por tanto, si `p` es un puntero persistente a la clase `T`, se puede usar la sintaxis estándar como `p->A` o `p->f(v)` para tener acceso al atributo `A` de la clase `T` o para invocar al método `f` de la clase `T`.

El sistema de bases de datos ObjectStore usa un enfoque diferente de los punteros persistentes. Utiliza los tipos normales de punteros para almacenar los punteros persistentes. Esto plantea dos problemas: (1) el tamaño de los punteros de la memoria sólo puede ser de 4 bytes, demasiado pequeño para las bases de datos mayores de 4 gigabytes, y (2) cuando se pasa algún objeto al disco los punteros de la memoria que señalan a su antigua ubicación física carecen de significado. ObjectStore usa una técnica denominada “rescate hardware” para abordar ambos problemas; precaptura los objetos de la base de datos en la memoria y sustituye los punteros persistentes por punteros de memoria y, cuando se vuelven a almacenar los datos en el disco, los punteros de memoria se sustituyen por punteros persistentes. Cuando se hallan en el disco, el valor almacenado

en el campo de los punteros de memoria no es el puntero persistente real; en vez de eso, se busca el valor en una tabla para averiguar el valor completo del puntero persistente.

- **Creación de objetos persistentes.** El operador `new` de C++ se usa para crear objetos persistentes mediante la definición de una versión “sobrecargada” del operador que usa argumentos adicionales especificando que deben crearse en la base de datos. Por tanto, en lugar de `new T()`, hay que llamar a `new (bd) T()` para crear un objeto persistente, donde `bd` identifica a la base de datos.
- **Extensiones de las clases.** Las extensiones de las clases se crean y se mantienen de manera automática para cada clase. La norma ODMG de C++ exige que el nombre de la clase se pase como parámetro adicional a la operación `new`. Esto también permite mantener varias extensiones para cada clase, pasando diferentes nombres.
- **Relaciones.** Las relaciones entre las clases se suelen representar almacenando punteros de cada objeto a los objetos con los que está relacionado. Los objetos relacionados con varios objetos de una clase dada almacenan un conjunto de punteros. Por tanto, si un par de objetos se halla en una relación, cada uno de ellos debe almacenar un puntero al otro. Los sistemas persistentes de C++ ofrecen una manera de especificar esas restricciones de integridad y de hacer que se cumplan mediante la creación y borrado automático de los punteros. Por ejemplo, si se crea un puntero de un objeto *a* a un objeto *b*, se añade de manera automática un puntero a *a* al objeto *b*.
- **Interfaz iteradora.** Dado que los programas tienen que iterar sobre los miembros de las clases, hace falta una interfaz para iterar sobre los miembros de las extensiones de las clases. La interfaz iteradora también permite especificar selecciones, de modo que sólo hay que capturar los objetos que satisfagan el predicado de selección.
- **Transacciones.** Los sistemas persistentes de C++ ofrecen soporte para comenzar las transacciones y para comprometerlas o provocar su retroceso.
- **Actualizaciones.** Uno de los objetivos de ofrecer soporte a la persistencia a los lenguajes de programación es permitir la persistencia transparente. Es decir, una función que opere sobre un objeto no debe necesitar saber que el objeto es persistente; por tanto, se pueden usar las mismas funciones sobre los objetos independientemente de que sean persistentes o no.

Sin embargo, surge el problema de que resulta difícil detectar cuándo se ha actualizado un objeto. Algunas extensiones persistentes de C++ exigían que el programador especificara de manera explícita que se ha modificado el objeto llamando a la función `mark_modified()`. Además de incrementar el esfuerzo del programador, este enfoque aumenta la posibilidad de que se produzcan errores de programación que den lugar a una base de datos corrupta. Si un programador omite la llamada a `mark_modified()`, es posible que nunca se propague a la base de datos la actualización llevada a cabo por una transacción, mientras que otra actualización realizada por la misma transacción sí se propaga, lo que viola la atomicidad de las transacciones.

Otros sistemas, como ObjectStore, usan soporte a la protección de la memoria proporcionada por el sistema operativo o por el hardware para detectar las operaciones de escritura en los bloques de memoria y marcar como sucios los bloques que se deban escribir posteriormente en el disco.

- **Lenguaje de consultas.** Los iteradores ofrecen soporte para consultas de selección sencillas. Para soportar consultas más complejas los sistemas persistentes de C++ definen un lenguaje de consultas.

Gran número de sistemas de bases de datos orientados a objetos basados en C++ se desarrollaron a finales de los años ochenta y principios de los noventa del siglo veinte. Sin embargo, el mercado para esas bases de datos resultó mucho más pequeño de lo esperado, ya que la mayor parte de los requisitos de las aplicaciones se cumplen de sobra usando SQL mediante interfaces como ODBC o JDBC. En consecuencia, la mayor parte de los sistemas de bases de datos orientados a objetos desarrollados en ese periodo ya no existen. En los años noventa el Grupo de Gestión de Datos de Objetos (Object Data Management Group, ODMG) definió las normas para agregar persistencia a C++ y a Java. No obstante,

el grupo concluyó sus actividades alrededor de 2002. ObjectStore y Versant son de los pocos sistemas de bases de datos orientados a objetos originales que siguen existiendo.

Aunque los sistemas de bases de datos orientados a objetos no encontraron el éxito comercial que esperaban, la razón de añadir persistencia a los lenguajes de programación sigue siendo válida. Hay varias aplicaciones con grandes exigencias de rendimiento que se ejecutan en sistemas de bases de datos orientados a objetos; el uso de SQL impondría una sobrecarga de rendimiento excesiva para muchos de esos sistemas. Con los sistemas de bases de datos relacionales orientados a objetos que proporcionan soporte para los tipos de datos complejos, incluidas las referencias, resulta más sencillo almacenar los objetos de los lenguajes de programación en bases de datos de SQL. Todavía puedeemerger una nueva generación de sistemas de bases de datos orientadas a objetos que utilicen bases de datos relacionales orientadas a objetos como sustrato.

9.8.5 Sistemas Java persistentes

En años recientes el lenguaje Java ha visto un enorme crecimiento en su uso. La demanda de soporte de la persistencia de los datos en los programas de Java se ha incrementado de manera acorde. Los primeros intentos de creación de una norma para la persistencia en Java fueron liderados por el consorcio ODMG; posteriormente, el consorcio concluyó sus esfuerzos, pero transfirió su diseño al proyecto **Objetos de bases de datos de Java** (Java Database Objects, JDO), que coordina Sun Microsystems.

El modelo JDO para la persistencia de los objetos en los programas de Java es diferente del modelo de soporte de la persistencia en los programas de C++. Entre sus características se hallan:

- **Persistencia por alcance.** Los objetos no se crean explícitamente en la base de datos. El registro explícito de un objeto como persistente (usando el método `makePersistent()` de la clase `PersistenceManager`) hace que el objeto sea persistente. Además, cualquier objeto alcanzable desde un objeto persistente pasa a ser persistente.
- **Mejora del código de bytes.** En lugar de declarar en el código de Java que una clase es persistente, se especifican en un archivo de configuración (con la extensión `.jdo`) las clases cuyos objetos se pueden hacer persistentes. Se ejecuta un programa *mejorador* específico de la implementación que lee el archivo de configuración y lleva a cabo dos tareas. En primer lugar, puede crear estructuras en la base de datos para almacenar objetos de esa clase. En segundo lugar, modifica el código de bytes (generado al compilar el programa de Java) para que maneje tareas relacionadas con la persistencia. A continuación se ofrecen algunos ejemplos de modificaciones de este tipo.
 - Se puede modificar cualquier código que tenga acceso a un objeto para que compruebe primero si el objeto se halla en la memoria y, si no está, dé los pasos necesarios para ponerlo en la memoria.
 - Cualquier código que modifique un objeto se modifica para que también registre el objeto como modificado y, quizás, para que guarde un valor previo a la actualización que se usa en caso de que haga falta deshacerla (es decir, si se retrocede la transacción).

También se pueden llevar a cabo otras modificaciones del código de bytes. Esas modificaciones del código de bytes son posibles porque el código de bytes es estándar en todas las plataformas e incluye mucha más información que el código objeto compilado.

- **Asignación de bases de datos.** JDO no define la manera en que se almacenan los datos en la base de datos subyacente. Por ejemplo, una situación frecuente es que los objetos se almacenen en una base de datos relacional. El programa mejorador puede crear en la base de datos un esquema adecuado para almacenar los objetos de las clases. La manera exacta en que lo hace depende de la implementación y no está definida por JDO. Se pueden asignar algunos atributos a los atributos relacionales, mientras que otros se pueden almacenar de forma serializada, que la base de datos trata como si fuera un objeto binario. Las implementaciones de JDO pueden permitir que los datos relacionales existentes se vean como objetos mediante la definición de la asignación correspondiente.
- **Extensiones de clase.** Las extensiones de clase se crean y se conservan de manera automática para cada clase declarada como persistente. Todos los objetos que se hacen persistentes se añaden

de manera automática a la extensión de clase correspondiente a su clase. Los programas de JDO pueden tener acceso a las extensiones de clase e iterar sobre los miembros seleccionados. La interfaz Iteradora ofrecida por Java se puede usar para crear iteradores sobre las extensiones de clase y avanzar por los miembros de cada extensión de clase. JDO también permite que se especifiquen selecciones cuando se crea una extensión de clase y que sólo se capturen los objetos que satisfagan la selección.

- **Tipo de referencia único.** No hay diferencia de tipos entre las referencias a los objetos transitorios y las referencias a los objetos persistentes.

Un enfoque para conseguir esta unificación de los tipos de puntero es cargar toda la base de datos en la memoria, lo que sustituye todos los punteros persistentes por punteros de memoria. Una vez llevadas a cabo las actualizaciones, el proceso se invierte y se vuelven a almacenar en el disco los objetos actualizados. Este enfoque es muy ineficiente para bases de datos de gran tamaño.

A continuación se describirá un enfoque alternativo, que permite que los objetos persistentes se capturen en memoria de manera automática cuando hace falta, mientras que permite que todas las referencias contenidas en objetos de la memoria sean referencias de la memoria. Cuando se captura el objeto A , se crea un **objeto hueco** para cada objeto B_i al que haga referencia, y la copia de la memoria de A tiene referencias al objeto hueco correspondiente para cada B_i . Por supuesto, el sistema tiene que asegurar que, si ya se ha capturado el objeto B_i , la referencia apunta al objeto ya capturado en lugar de crear un nuevo objeto hueco. De manera parecida, si no se ha capturado el objeto B_i , pero otro objeto ya capturado hace referencia a él, ya tiene un objeto hueco creado para él; la referencia al objeto hueco ya existente se vuelve a usar, en lugar de crear un objeto hueco nuevo.

Por tanto, para cada objeto O_i que se haya capturado, cada referencia de O_i es a un objeto ya capturado o a un objeto hueco. Los objetos huecos forman un *borde* que rodea los objetos capturados.

Siempre que el programa tiene acceso real al objeto hueco O , el código de bytes mejorado lo detecta y captura el objeto en la base de datos. Cuando se captura este objeto, se lleva a cabo el mismo proceso de creación de objetos huecos para todos los objetos a los que O hace referencia. Tras esto, se permite que se lleve a cabo el acceso al objeto³.

Para implementar este esquema hace falta una estructura de índices en la memoria que asigne los punteros persistentes a las referencias de la memoria. Cuando se vuelven a escribir los objetos en el disco se usa ese índice para sustituir las referencias de la memoria por punteros persistentes en la copia que se escribe en el disco.

La norma JDO se halla todavía en una etapa preliminar y sometida a revisión. Varias empresas ofrecen implementaciones de JDO. No obstante, queda por ver si JDO se usará mucho, a diferencia de C++ de ODMG.

9.9 Sistemas orientados a objetos y sistemas relacionales orientados a objetos

Ya se han estudiado las bases de datos relacionales orientadas a objetos, que son bases de datos orientadas a objetos construidas sobre el modelo relacional, así como las bases de datos orientadas a objetos, que se crean alrededor de los lenguajes de programación persistentes.

3. La técnica que usa objetos huecos descrita anteriormente se halla estrechamente relacionada con la técnica de rescate hardware (ya mencionada en el Apartado 9.8.4). El rescate hardware la usan algunas implementaciones persistentes de C++ para ofrecer un solo tipo de puntero para los punteros persistentes y los de memoria. El rescate hardware utiliza técnicas de protección de la memoria virtual proporcionadas por el sistema operativo para detectar el acceso a las páginas y captura las páginas de la base de datos cuando es necesario. Por el contrario, la versión de Java modifica el código de bytes para que compruebe la existencia de objetos huecos en lugar de usar la protección de la memoria y captura los objetos cuando hace falta, en vez de capturar páginas enteras de la base de datos.

Las extensiones persistentes de los lenguajes de programación y los sistemas relationales orientados a objetos se dirigen a mercados diferentes. La naturaleza declarativa y la limitada potencia (comparada con la de los lenguajes de programación) del lenguaje SQL proporcionan una buena protección de los datos respecto de los errores de programación y hacen que las optimizaciones de alto nivel, como la reducción de E/S, resulten relativamente sencillas (la optimización de las expresiones relationales se trata en el Capítulo 14). Los sistemas relationales orientados a objetos se dirigen a simplificar la realización de los modelos de datos y de las consultas mediante el uso de tipos de datos complejos. Entre las aplicaciones habituales están el almacenamiento y la consulta de datos complejos, incluidos los datos multimedia.

Los lenguajes declarativos como SQL, sin embargo, imponen una reducción significativa del rendimiento a ciertos tipos de aplicaciones que se ejecutan principalmente en la memoria principal y realizan gran número de accesos a la base de datos. Los lenguajes de programación persistentes se dirigen a las aplicaciones de este tipo que tienen necesidad de un rendimiento elevado. Proporcionan acceso a los datos persistentes con poca sobrecarga y eliminan la necesidad de traducir los datos si hay que tratarlos con un lenguaje de programación. Sin embargo, son más susceptibles de deteriorar los datos debido a los errores de programación y no suelen disponer de gran capacidad de consulta. Entre las aplicaciones habituales están las bases de datos de CAD.

Los puntos fuertes de los diversos tipos de sistemas de bases de datos pueden resumirse de la manera siguiente:

- **Sistemas relationales:** tipos de datos sencillos, lenguajes de consultas potentes, protección elevada.
- **Bases de datos orientadas a objetos basadas en lenguajes de programación persistentes:** tipos de datos complejos, integración con los lenguajes de programación, elevado rendimiento.
- **Sistemas relationales orientados a objetos:** tipos de datos complejos, lenguajes de consultas potentes, protección elevada.

Estas descripciones son válidas en general, pero hay que tener en cuenta que algunos sistemas de bases de datos no respetan estas fronteras. Por ejemplo, algunos sistemas de bases de datos orientados a objetos construidos alrededor de lenguajes de programación persistentes se pueden implementar sobre sistemas de bases de datos relationales o sobre sistemas de bases de datos relationales orientados a objetos. Puede que estos sistemas proporcionen menor rendimiento que los sistemas de bases de datos orientados a objetos construidos directamente sobre los sistemas de almacenamiento, pero proporcionan parte de las garantías de protección más estrictas propias de los sistemas relationales.

9.10 Resumen

- El modelo de datos relacional orientado a objetos extiende el modelo de datos relacional al proporcionar un sistema de tipos enriquecido que incluye los tipos de conjuntos y la orientación a objetos.
- Los tipos de conjuntos incluyen las relaciones anidadas, los conjuntos, los multiconjuntos y los arrays, y el modelo relacional orientado a objetos permite que los atributos de las tablas sean conjuntos.
- La orientación a objetos proporciona herencia con subtipos y subtablas, así como referencias a objetos (tuplas).
- La norma SQL:1999 extiende el lenguaje de definición de datos y de consultas con los nuevos tipos de datos y la orientación a objetos.
- Se han estudiado varias características del lenguaje de definición de datos extendido, así como del lenguaje de consultas y, en especial, da soporte a los atributos valorados como conjuntos, a la herencia y a las referencias a las tuplas. Estas extensiones intentan conservar los fundamentos relationales—en particular, el acceso declarativo a los datos—a la vez que se extiende la potencia de modelado.

- Los sistemas de bases de datos relacionales orientados a objetos (es decir, los sistemas de bases de datos basados en el modelo relacional orientado a objetos) ofrecen un camino de migración cómodo para los usuarios de las bases de datos relacionales que desean usar las características orientadas a objetos.
- Las extensiones persistentes de C++ y Java integran la persistencia de forma elegante y orthogonalmente a sus elementos de programación previos, por lo que resulta fácil de usar.
- La norma ODMG define las clases y otros constructores para la creación y acceso a los objetos persistentes desde C++, mientras que la norma JDO ofrece una funcionalidad equivalente para Java.
- Se han estudiado las diferencias entre los lenguajes de programación persistentes y los sistemas relacionales orientados a objetos, y se han mencionado criterios para escoger entre ellos.

Términos de repaso

- Relaciones anidadas.
- Modelo relacional anidado.
- Tipos complejos.
- Tipos de conjuntos.
- Tipos de objetos grandes.
- Conjuntos.
- Arrays.
- Multiconjuntos.
- Tipos estructurados.
- Métodos.
- Tipos fila.
- Funciones constructoras.
- Herencia:
 - Simple.
 - Múltiple.
- Herencia de tipos.
- Tipo más concreto.
- Herencia de tablas.
- Subtabla.
- Solapamiento de subtablas.
- Tipos de referencia.
- Ámbito de una referencia.
- Atributo autorreferencial.
- Expresiones de camino.
- Anidamiento y desanidamiento.
- Funciones y procedimientos en SQL.
- Lenguajes de programación persistentes.
- Persistencia por:
 - Clase.
 - Creación.
 - Marcado.
 - Alcance.
- Enlace C++ de ODMG.
- ObjectStore.
- JDO
 - Persistencia por alcance.
 - Raíces.
 - Objetos huecos.
 - Asignación relacional de objetos.

Ejercicios prácticos

9.1 Una compañía de alquiler de coches tiene una base de datos con todos los vehículos de su flota actual. Para todos los vehículos incluye el número de bastidor, el número de matrícula, el fabricante, el modelo, la fecha de adquisición y el color. Se incluyen datos especiales para algunos tipos de vehículos:

- Camiones: capacidad de carga.
- Coches deportivos: potencia, edad mínima del arrendatario.
- Furgonetas: número de plazas.
- Vehículos todoterreno: altura de los bajos, eje motor (tracción a dos ruedas o a las cuatro).

Constrúyase una definición del esquema de esta base de datos de acuerdo con SQL:1999. Utilícese la herencia donde resulte conveniente.

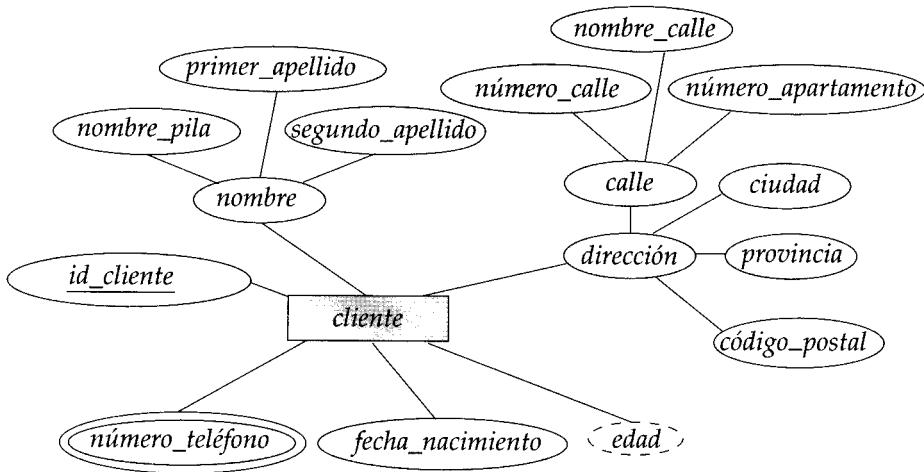


Figura 9.5 Diagrama E-R con atributos compuestos, multivalorados y derivados.

9.2 Considérese el esquema de la base de datos con la relación *Emp* cuyos atributos se muestran a continuación, con los tipos especificados para los atributos multivalorados.

Emp = (*nombre*, ConjuntoHijos **multiset**(*HijosC*), ConjuntoConocimientos **multiset**(*Conocimientos*))
Hijos = (*nombre*, *cumpleaños*)

Conocimientos = (*mecanografía*, ConjuntoExámenes **setof**(*Exámenes*))

Exámenes = (*año*, *ciudad*)

- a. Definir el esquema anterior en SQL:2003, con los tipos correspondientes para cada atributo.
- b. Usando el esquema anterior, escribir las consultas siguientes en SQL:2003.
 - I. Averiguar el nombre de todos los empleados que tienen un hijo nacido el 1 de enero de 2000 o en fechas posteriores.
 - II. Averiguar los empleados que han hecho un examen del tipo de conocimiento “mecanografía” en la ciudad “San Rafael”.
 - III. Hacer una relación de los tipos de conocimiento de la relación *Emp*.

9.3 Considérese el diagrama E-R de la Figura 9.5, que contiene atributos compuestos, multivalorados y derivados.

- a. Dese una definición de esquema en SQL:2003 correspondiente al diagrama E-R.
- b. Dense constructores para cada uno de los tipos estructurados definidos.

9.4 Considérese el esquema relacional de la Figura 9.6.

- a. Dese una definición de esquema en SQL:2003 correspondiente al esquema relacional, pero usando referencias para expresar las relaciones de clave externa.
- b. Escríbanse cada una de las consultas del Ejercicio 2.9 sobre el esquema anterior usando SQL:2003.

9.5 Supóngase que se trabaja como asesor para escoger un sistema de bases de datos para la aplicación del cliente. Para cada una de las aplicaciones siguientes indíquese el tipo de sistema de bases de datos (relacional, base de datos orientada a objetos basada en un lenguaje de programación per-

empleado (nombre_empleado, *calle*, *ciudad*)
trabaja (nombre_empleado, nombre_empresa, *sueldo*)
empresa (nombre_empresa, *ciudad*)
supervisa (nombre_empleado, nombre_jefe)

Figura 9.6 Base de datos relacional para el Ejercicio práctico 9.4.

sistente, relacional orientada a objetos; no se debe especificar ningún producto comercial) que se recomendaría. Justifíquese la recomendación.

- a. Sistema de diseño asistido por computadora para un fabricante de aviones.
- b. Sistema para realizar el seguimiento de los donativos hechos a los candidatos a un cargo público
- c. Sistema de información de ayuda para la realización de películas.

9.6 ¿En qué se diferencia el concepto de objeto del modelo orientado a objetos del concepto de entidad del modelo entidad-relación?

Ejercicios

- 9.7 Vuélvase a diseñar la base de datos del Ejercicio práctico 9.2 en la primera y en la cuarta formas normales. Indíquense las dependencias funcionales o multivaloradas que se den por supuestas. Indíquense también todas las restricciones de integridad referencial que deban incluirse en los esquemas de la primera y de la cuarta formas normales.
- 9.8 Considérese el esquema del Ejercicio práctico 9.2.
- a. Dense instrucciones del LDD de SQL:2003 para crear la relación *EmpA*, que tiene la misma información que *Emp*, pero en la que los atributos valorados como multiconjuntos *ConjuntoNiños*, *ConjuntoConocimientos* y *ConjuntoExámenes* se sustituyen por los atributos valorados como arrays *ArrayHijos*, *ArrayConocimientos* y *ArrayExámenes*.
 - b. Escríbase una consulta para transformar los datos del esquema de *Emp* al de *EmpA*, con el array de los hijos ordenado por fecha de cumpleaños, el de conocimientos por el tipo de conocimientos y el de exámenes por el año de realización.
 - c. Escríbase una instrucción de SQL para actualizar la relación *Emp* añadiendo el hijo Jorge, con fecha de nacimiento de 5 de febrero de 2001 al empleado llamado Gabriel.
 - d. Escríbase una instrucción de SQL para llevar a cabo la misma actualización que antes, pero sobre la relación *EmpA*. Asegúrese de que el array de los hijos sigue ordenado por años.
- 9.9 Considérense los esquemas de la tabla *personas* y las tablas *estudiantes* y *profesores* que se crearon bajo *personas* en el Apartado 9.4. Dese un esquema relacional en la tercera forma normal que represente la misma información. Recuérdense las restricciones de las subtablas y dense todas las restricciones que deban imponerse en el esquema relacional para que cada ejemplar de la base de datos del esquema relacional pueda representarse también mediante un ejemplar del esquema con herencia.
- 9.10 Explíquese la diferencia entre el tipo *x* y el tipo de referencia *ref(x)*. ¿En qué circunstancias se debe escoger usar el tipo de referencia?
- 9.11
- a. Dese una definición de esquema en SQL:1999 del diagrama E-R de la Figura 9.7, que contiene especializaciones, usando subtipos y subtablas.
 - b. Dese una consulta de SQL:1999 para averiguar el nombre de todas las personas que no son secretarias.
 - c. Dese una consulta de SQL:1999 para imprimir el nombre de las personas que no sean empleados ni clientes.
 - d. ¿Se puede crear una persona que sea a la vez empleado y cliente con el esquema que se acaba de crear? Explíquese la manera de hacerlo o el motivo de que no sea posible.
- 9.12 Supóngase que una base de datos de JDO tiene un objeto *A*, que hace referencia al objeto *B*, que, a su vez, hace referencia al objeto *C*. Supóngase que todos los objetos se hallan inicialmente en el disco. Supóngase que un programa desvincula en primer lugar a *A*, luego a *B* siguiendo la referencia de *A* y, finalmente, desvincula a *C*. Muéstrense los objetos que están en representados en memoria tras cada desvinculación, junto con su estado (hueco o lleno, y los valores de los campos de referencia).

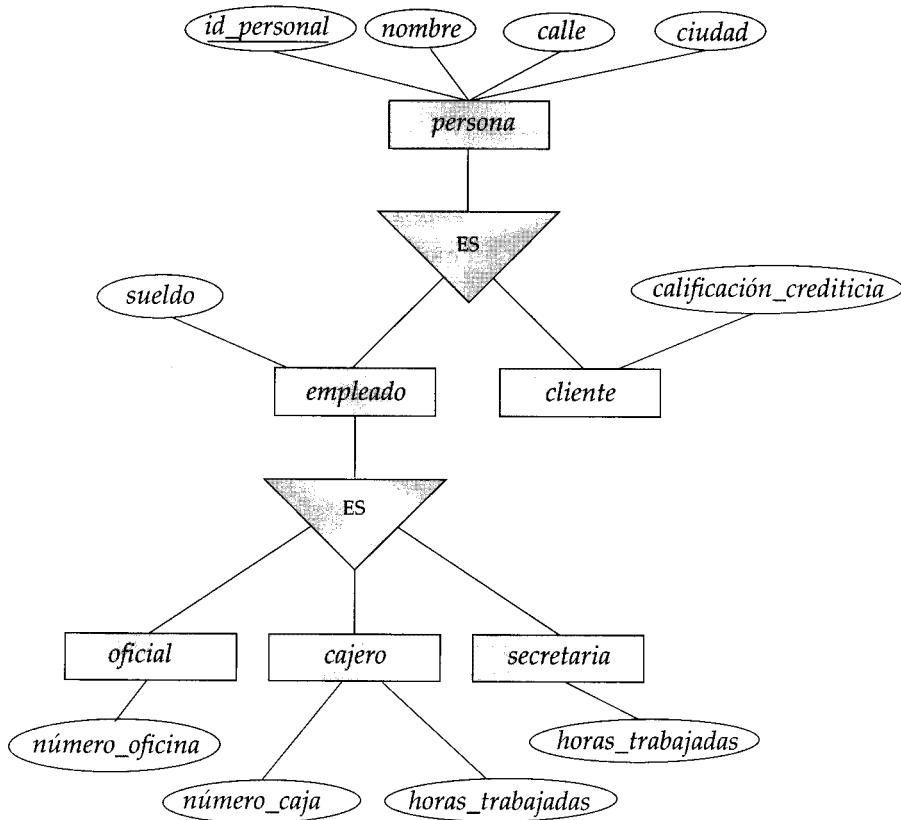


Figura 9.7 Especialización y generalización.

Notas bibliográficas

Se han propuesto varias extensiones de SQL orientadas a objetos. POSTGRES (Stonebraker y Rowe [1986] y Stonebraker [1986]) fue una de las primeras implementaciones de un sistema relacional orientado a objetos. Otros sistemas relacionales orientados a objetos de la primera época son las extensiones de SQL de *O₂* (Bancilhon et al. [1989]) y UniSQL (UniSQL [1991]). SQL:1999 fue producto de un amplio (y muy tardío) esfuerzo de normalización, que se inició originalmente mediante el añadido a SQL de características orientadas a objetos y acabó añadiendo muchas más características, como las estructuras procedimentales que se han visto anteriormente. El soporte de los tipos multiconjuntos se añadió como parte de SQL:2003.

Entre los libros de texto sobre SQL:1999 están Melton y Simon [2001] y Melton [2002]; el último se centra en las características relacionales orientadas a objetos de SQL:1999. Eisenberg et al. [2004] ofrece una visión general de SQL:2003, incluido el soporte de los multiconjuntos. Se deben consultar los manuales (en línea) del sistema de bases de datos que se utilice para averiguar las características de SQL:1999 y de SQL:2003 que soporta.

A finales de los años ochenta y principios de los años noventa del siglo veinte se desarrollaron varios sistemas de bases de datos orientadas a objetos. Entre los más notables de los comerciales figuran ObjectStore (Lamb et al. [1991]), *O₂* (Lecluse et al. [1988]) y Versant. La norma para bases de datos orientadas a objetos ODMG se describe con detalle en Cattell [2000]. JDO se describe en Roos [2002], Tyagi et al. [2003] y Jordan y Russell [2003].

Herramientas

Hay diferencias considerables entre los diversos productos de bases de datos en cuanto al soporte de las características relacionales orientadas a objetos. Probablemente Oracle tenga el soporte más amplio entre los principales fabricantes de bases de datos. El sistema de bases de datos de Informix ofrece so-

porte para muchas características relacionales orientadas a objetos. Tanto Oracle como Informix ofrecían características relacionales orientadas a objetos antes de la finalización de la norma SQL:1999 y presentan algunas características que no forman parte de SQL:1999.

La información sobre ObjectStore y sobre Versant, incluida la descarga de versiones de prueba, se puede obtener de sus respectivos sitios Web (objectstore.com y versant.com). El proyecto Apache DB (db.apache.org) ofrece una herramienta de asignación relacional orientada a objetos para Java que soporta tanto Java de ODMG como las APIs de JDO. Se puede obtener una implementación de referencia de JDO de sun.com; se puede usar un motor de búsqueda para averiguar el URL completo.

XML

A diferencia de la mayor parte de las tecnologías presentadas en los capítulos anteriores, el **lenguaje de marcas extensible** (Extensible Markup Language, XML) no se concibió como una tecnología para bases de datos. En realidad, al igual que el *lenguaje de marcas de hipertexto* (*Hyper-Text Markup Language*, HTML) en el que está basado la World Wide Web, XML tiene sus raíces en la gestión de documentos y está derivado de un lenguaje para estructurar documentos grandes conocido como *lenguaje estándar generalizado de marcas* (*Standard Generalized Markup Language*, SGML). Sin embargo, a diferencia de SGML y de HTML, XML puede representar datos de bases de datos, así como muchas clases de datos estructurados. Es particularmente útil como formato de datos cuando las aplicaciones se deben comunicar con otra aplicación o integrar información de varias aplicaciones. Cuando XML se usa en estos contextos, se generan muchas dudas sobre las bases de datos, incluyendo cómo organizar, manipular y consultar los datos XML. En este capítulo se introduce XML y se estudia la gestión de los datos XML con las técnicas de bases de datos, así como el intercambio de datos con formato como documentos XML.

10.1 Motivación

Para comprender XML es importante entender sus raíces como un lenguaje de marcas de documentos. El término **marca** se refiere a cualquier elemento en un documento del que no se tiene intención que sea parte de la salida impresa. Por ejemplo, un escritor que crea un texto que finalmente se compone en una revista puede desear realizar notas sobre cómo se ha de realizar la composición. Sería importante introducir estas notas de tal forma que se pudieran distinguir del contenido real; una nota como “usar un tipo mayor y poner en negrita” o “no romper este párrafo” no acabaría impresa en la revista. Estas notas comunican información adicional sobre el texto. En un procesamiento electrónico de documentos un **lenguaje de marcas** es una descripción formal de la parte del documento que es contenido, la parte que es marca y lo que significa la marca.

Así como los sistemas de bases de datos evolucionaron desde el procesamiento físico de archivos para proporcionar una vista lógica aislada, los lenguajes de marcas evolucionaron desde la especificación de instrucciones que indicaban cómo imprimir partes del documento para la *función* del contenido. Por ejemplo, con marcas funcionales, el texto que representa los encabezamientos de sección (para este apartado, la palabra “Motivación”) se marcaría como un encabezamiento de apartado en lugar de marcarse como un texto con el fin de imprimirse en tamaño grande y negrita. Desde el punto de vista del editor, dicha marca funcional permite que el documento tenga distintos formatos en contextos diferentes. También ayuda a que distintas partes de un documento largo, o distintas páginas en un sitio Web grande, tengan un formato uniforme. Más importante, la marca funcional también ayuda a registrar lo que representa semánticamente cada parte del texto y ayuda a la extracción automática de partes claves de los documentos.

Para la familia de lenguajes de marcado, en los que se incluyen HTML, SGML y XML, las marcas adoptan la forma de **etiquetas** encerradas entre corchetes angulares, <>. Las etiquetas se usan en pares, con

```

<banco>
  <cuenta>
    <número_cuenta> C-101 </número_cuenta>
    <nombre_sucursal> Centro </nombre_sucursal>
    <saldo> 500 </saldo>
  </cuenta>
  <cuenta>
    <número_cuenta> C-102 </número_cuenta>
    <nombre_sucursal> Navacerrada </nombre_sucursal>
    <saldo> 400 </saldo>
  </cuenta>
  <cuenta>
    <número_cuenta> C-201 </número_cuenta>
    <nombre_sucursal> Galapagar </nombre_sucursal>
    <saldo> 900 </saldo>
  </cuenta>
  <cliente>
    <nombre_cliente> González </nombre_cliente>
    <calle_cliente> Arenal </calle_cliente>
    <ciudad_cliente> La Granja </ciudad_cliente>
  </cliente>
  <cliente>
    <nombre_cliente> López </nombre_cliente>
    <calle_cliente> Mayor </calle_cliente>
    <ciudad_cliente> Peguerinos </ciudad_cliente>
  </cliente>
  <impositor>
    <número_cuenta> C-101 </número_cuenta>
    <nombre_cliente> González </nombre_cliente>
  </impositor>
  <impositor>
    <número_cuenta> C-201 </número_cuenta>
    <nombre_cliente> González </nombre_cliente>
  </impositor>
  <impositor>
    <número_cuenta> C-102 </número_cuenta>
    <nombre_cliente> López </nombre_cliente>
  </impositor>
</banco>

```

Figura 10.1 Representación XML de información bancaria.

<etiqueta> y </etiqueta> delimitando el comienzo y final de la porción de documento a la cual se refiere la etiqueta. Por ejemplo, el título de un documento podría estar marcado de la siguiente forma:

```
<title>Fundamentos de bases de datos</title>
```

A diferencia de HTML, XML no impone las etiquetas permitidas, y se pueden elegir como sea necesario para cada aplicación. Esta característica es la clave de la función principal de XML en la representación e intercambio de datos, mientras que HTML se usa principalmente para el formato de documentos.

Por ejemplo, en nuestra aplicación del banco, la información de la cuenta y del cliente se puede representar como parte de un documento XML, como se muestra en la Figura 10.1. Obsérvese el uso de etiquetas tales como cuenta y número_cuenta. Estas etiquetas proporcionan el contexto de cada valor y permiten identificar la semántica del valor. Para este ejemplo, la representación de datos XML no

proporciona ninguna ventaja significativa con respecto a la representación relacional; sin embargo, se usa este ejemplo por su simplicidad.

La Figura 10.2, que muestra la forma en que se puede representar un pedido de compra en XML, ilustra un uso más realista de XML. Los pedidos de compra se generan habitualmente por una empresa y se envían a otra. Tradicionalmente el comprador los imprimía en papel y los enviaba al proveedor; éste reintroducía manualmente los datos en el sistema informático. Este lento proceso se puede acelerar en gran medida enviando electrónicamente la información entre el comprador y el proveedor. La representación anidada permite que toda la información de un pedido de compra se represente de forma natural en un único documento (los pedidos de compra reales tienen considerablemente más información que la que se muestra en este ejemplo simple). XML proporciona una forma estándar de etiquetar los datos; obviamente, las dos empresas deben estar de acuerdo en las etiquetas que aparecen en el pedido de compra y lo que significan.

Comparado con el almacenamiento de los datos en una base de datos relacional, la representación XML puede parecer poco eficiente, puesto que los nombres de las etiquetas se repiten por todo el documento. Sin embargo, a pesar de esta desventaja, una representación XML presenta ventajas significativas cuando se usa para el intercambio de datos entre empresas y para almacenar información estructurada en archivos.

- En primer lugar la presencia de las etiquetas hace que el mensaje sea **autodocumentado**, es decir, no se tiene que consultar un esquema para comprender el significado del texto. Por ejemplo, se puede leer fácilmente el fragmento anterior.
- En segundo lugar, el formato del documento no es rígido. Por ejemplo, si algún remitente agrega información adicional tal como una etiqueta `último_acceso` que informa de la última fecha en la que se ha accedido a la cuenta, el recipiente de los datos XML puede sencillamente ignorar la etiqueta. Como ejemplo adicional, en la Figura 10.2, el elemento con el identificador SG2 tiene una etiqueta denominada `unidad_de_medida`, que el primer elemento no tiene. La etiqueta se requiere en los elementos que se ordenan por peso o volumen, y se puede omitir en elementos que simplemente se ordenan por número.

La capacidad de reconocer e ignorar las etiquetas inesperadas permite al formato de los datos evolucionar con el tiempo sin invalidar las aplicaciones existentes. De forma similar, la posibilidad de que la misma etiqueta aparezca varias veces hace que sea fácil representar atributos multivalorados.

- En tercer lugar, XML permite estructuras anidadas. El pedido de compra mostrado en la Figura 10.2 ilustra las ventajas de tener estas estructuras. Cada pedido de compra tiene un comprador y una lista de elementos como dos de sus estructuras anidadas. Cada elemento tiene a su vez un identificador, una descripción y un precio anidados, mientras que el comprador tiene un nombre y dirección anidados.

Esta información se podría haber dividido en varias relaciones en el modelo relacional. La información de los elementos se habría almacenado en una relación, la información del comprador en una segunda relación, los pedidos de compra en una tercera y la relación entre los pedidos de compra, compradores y elementos en una cuarta.

La representación relacional ayuda a evitar la redundancia; por ejemplo, las descripciones de los elementos se almacenarían sólo una vez por cada identificador de elemento en un esquema relacional normalizado. Sin embargo, en el pedido de compra de XML, las descripciones se pueden repetir en varios pedidos con el mismo elemento. No obstante, es atractivo recoger toda la información relacionada con un pedido en una única estructura anidada, incluso añadiendo redundancia, cuando la información se tiene que intercambiar con terceros.

- Finalmente, puesto que el formato XML está ampliamente aceptado hay una gran variedad de herramientas disponibles para ayudar a su procesamiento, incluyendo APIs para lenguajes de programación para crear y leer datos XML, software de exploración y herramientas de bases de datos.

```

<pedido_compra>
    <identificador> P-101 </identificador>
    <comprador>
        <nombre> Coyote Loco </nombre>
        <dirección> Mesa Flat, Route 66, Arizona 12345, EEUU</dirección>
    </comprador>
    <proveedor>
        <nombre> Proveedores Acme SA</nombre>
        <dirección> 1, Broadway, Nueva York, NY, EEUU</dirección>
    </proveedor>
    <lista_elementos>
        <elemento>
            <identificador>EA1</identificador>
            <descripción> Trineo propulsado por cohetes atómicos</descripción>
            <cantidad> 2 </cantidad>
            <precio> 199.95 </precio>
        </elemento>
        <elemento>
            <identificador>PF2</identificador>
            <descripción> Pegamento fuerte</descripción>
            <cantidad> 1 </cantidad>
            <unidad_de_medida> litro </unidad_de_medida>
            <precio> 29.95 </precio>
        </elemento>
    </lista_elementos>
    <coste_total> 429.85 </coste_total>
    <forma_de_pago> Reembolso </forma_de_pago>
    <forma_de_envío> Avión </forma_de_envío>
</pedido_compra>

```

Figura 10.2 Representación XML de un pedido de compra.

Más adelante se describen varias aplicaciones de XML en la sección 10.7. Al igual que SQL es el *lenguaje* dominante para consultar los datos relacionales, XML se está convirtiendo en el *formato* dominante para el intercambio de datos.

10.2 Estructura de los datos XML

El constructor fundamental en un documento XML es el **elemento**. Un elemento es sencillamente un par de etiquetas de inicio y finalización coincidentes y todo el texto que aparece entre ellas. Los documentos XML deben tener un único elemento **raíz** que abarque al resto de elementos en el documento. En el ejemplo de la Figura 10.1 el elemento `<banco>` forma el elemento raíz. Además, los elementos en un documento XML se deben **anidar** adecuadamente. Por ejemplo:

```

...
<cuenta>
    Esta cuenta se usa muy rara vez, por no decir nunca.
    <número_cuenta> C-102 </número_cuenta>
    <nombre_sucursal> Navacerrada </nombre_sucursal>
    <saldo> 400 </saldo>
</cuenta>
...

```

Figura 10.3 Mezcla de texto con subelementos.

```
<cuenta> ... <saldo> ... </saldo> ... </cuenta>
```

está anidado adecuadamente, mientras que

```
<cuenta> ... <saldo> ... </cuenta> ... </saldo>
```

no está adecuadamente anidado.

Aunque el anidamiento adecuado es una propiedad intuitiva, es necesario definirla más formalmente. Se dice que el texto aparece **en el contexto de** un elemento si aparece entre la etiqueta de inicio y la etiqueta de finalización de dicho elemento. Las etiquetas están anidadas adecuadamente si toda etiqueta de inicio tiene una única etiqueta de finalización coincidente que está en el contexto del mismo elemento padre.

Obsérvese que el texto puede estar mezclado con los subelementos de otro elemento, como en la Figura 10.3. Como con otras características de XML, esta libertad tiene más sentido en un contexto de procesamiento de documentos que en el contexto de procesamiento de datos y no es particularmente útil para representar en XML datos más estructurados, como es el contenido de las bases de datos.

La capacidad de anidar elementos con otros elementos proporciona una forma alternativa de representar información. La Figura 10.4 muestra una representación de la información bancaria de la Figura 10.1, pero con los elementos **cuenta** anidados con los elementos **cliente**, aunque almacenaría elementos **cuenta** de una forma redundante si pertenecen a varios clientes.

Las representaciones anidadas se usan ampliamente en las aplicaciones de intercambio de datos XML para evitar las reuniones. Por ejemplo, un pedido de compra almacenaría la dirección completa del emisor y receptor de una forma redundante en varios pedidos de compra, mientras que una representación normalizada puede requerir una reunión de registros de pedidos de compras con una relación *dirección_empresa* para obtener la información de la dirección.

```
<banco-1>
  <cliente>
    <número_cliente> González </número_cliente>
    <calle_cliente> Arenal </calle_cliente>
    <ciudad_cliente> La Granja </ciudad_cliente>
    <cuenta>
      <número_cuenta> C-101 </número_cuenta>
      <nombre_sucursal> Centro </nombre_sucursal>
      <saldo> 500 </saldo>
    </cuenta>
    <cuenta>
      <número_cuenta> C-201 </número_cuenta>
      <nombre_sucursal> Galapagar </nombre_sucursal>
      <saldo> 900 </saldo>
    </cuenta>
  </cliente>
  <cliente>
    <número_cliente> López </número_cliente>
    <calle_cliente> Mayor </calle_cliente>
    <ciudad_cliente> Peguerinos </ciudad_cliente>
    <cuenta>
      <número_cuenta> C-102 </número_cuenta>
      <nombre_sucursal> Navacerrada </nombre_sucursal>
      <saldo> 400 </saldo>
    </cuenta>
  </cliente>
</bano-1>
```

Figura 10.4 Representación XML anidada de información bancaria.

```

    ...
    <cuenta tipo_cuenta= "corriente">
        <número_cuenta> C-102 </número_cuenta>
        <nombre_sucursal> Navacerrada </nombre_sucursal>
        <saldo> 400 </saldo>
    </cuenta>
    ...

```

Figura 10.5 Uso de atributos.

Además de los elementos, XML especifica la noción de **atributo**. Por ejemplo, el tipo de una cuenta se puede representar como un atributo, como en la Figura 10.5. Los atributos de un elemento aparecen como pares *nombre=valor* antes del cierre “>” de una etiqueta. Los atributos son cadenas y no contienen marcas. Además, los atributos pueden aparecer solamente una vez en una etiqueta dada, al contrario que los subelementos, que pueden estar repetidos.

Obsérvese que en un contexto de construcción de un documento es importante la distinción entre subelemento y atributo (un atributo es implícitamente texto que no aparece en el documento impreso o visualizado). Sin embargo, en las aplicaciones de bases de datos y de intercambio de datos de XML esta distinción es menos relevante y la elección de representar los datos como un atributo o un subelemento es frecuentemente arbitraria.

Una nota sintáctica final es que un elemento de la forma `<elemento></elemento>`, que no contiene subelementos o texto, se puede abbreviar como `<elemento/>`; los elementos abreviados pueden, no obstante, contener atributos.

Puesto que los documentos XML se diseñan para su intercambio entre aplicaciones se tiene que introducir un mecanismo de **espacio de nombres** para permitir a las organizaciones especificar nombre únicos globalmente para que se usen como marcas de elementos en los documentos. La idea de un espacio de nombres es anteponer cada etiqueta o atributo con un identificador de recursos universal (por ejemplo, una dirección Web). Por ello, por ejemplo, si Banco Principal desea asegurar que los documentos XML creados no duplican las etiquetas usadas por los documentos de otros socios del negocio, se puede anteponer un identificador único con dos puntos a cada nombre de etiqueta. El banco puede usar un URL Web como el siguiente

<http://www.BancoPrincipal.com>

como un identificador único. El uso de identificadores únicos largos en cada etiqueta puede ser poco conveniente por lo que el espacio de nombres estándar proporciona una forma de definir una abreviatura para los identificadores.

En la Figura 10.6 el elemento raíz (`banco`) tiene un atributo `xmlns:BP`, que declara que BP está definido como abreviatura del URL dado anteriormente. Se puede usar entonces la abreviatura en varias marcas de elementos como se ilustra en la figura.

Un documento puede tener más de un espacio de nombres, declarado como parte del elemento raíz. Se puede entonces asociar elementos diferentes con espacios de nombres distintos. Se puede definir un *espacio de nombres predeterminado* mediante el uso del atributo `xmlns` en lugar de `xmlns:BP` en el elemento raíz. Los elementos sin un prefijo de espacio de nombres explícito pertenecen entonces al espacio de nombres predeterminado.

```

<banco xmlns:BP="http://www.BancoPrincipal.com">
    ...
    <BP:sucursal>
        <BP:nombre_sucursal> Centro </BP:nombre_sucursal>
        <BP:ciudad_sucursal> Arganzuela </BP:ciudad_sucursal>
    </BP:sucursal>
    ...
</banco>

```

Figura 10.6 Nombres únicos de etiqueta mediante el uso de espacios de nombres.

Algunas veces es necesario almacenar valores que contienen etiquetas sin que se interpreten como etiquetas XML. Para ello, XML permite esta construcción:

```
<![CDATA[<cuenta> ... </cuenta>]]>
```

Debido a que el texto `<cuenta>` está encerrado en CDATA, se trata como datos de texto normal, no como una etiqueta. El término CDATA viene de datos de tipo carácter (“character data” en inglés).

10.3 Esquema de los documentos XML

Las bases de datos contienen esquemas que se usan para restringir qué información se puede almacenar en la base de datos y para restringir los tipos de datos de la información almacenada. En cambio, los documentos XML se pueden crear de forma predeterminada sin un esquema asociado. Un elemento puede tener entonces cualquier subelemento o atributo. Aunque dicha libertad puede ser aceptable algunas veces dada la naturaleza autodescriptiva del formato de datos, no es útil generalmente cuando los documentos XML se deben procesar automáticamente como parte de una aplicación o incluso cuando se van a dar formato en XML a grandes cantidades de datos relacionados. Aquí se describirá el primer lenguaje de definición de esquemas incluido como parte de la norma XML, la definición de tipos de documentos (*Document Type Definition*), así como su sustituto *XML Schema*, definido más recientemente. También se usa otro lenguaje de definición de esquemas denominado Relax NG, pero no se estudia aquí; para obtener más información sobre Relax NG, consultense las referencias en el apartado de notas bibliográficas.

10.3.1 Definición de tipos de documentos

La **definición de tipos de documentos** (*Document Type Definition*, DTD) es una parte opcional de un documento XML. El propósito principal de DTD es similar al de un esquema: restringir el tipo de información presente en el documento. Sin embargo, DTD no restringe en realidad los tipos en el sentido de tipos básicos como entero o cadena. En su lugar solamente restringe el aspecto de subelementos y atributos en un elemento. DTD es principalmente una lista de reglas que indican el patrón de subelementos que aparecen en un elemento. La Figura 10.7 muestra una parte de una DTD de ejemplo de un documento de información bancaria; el documento XML de la Figura 10.1 se ajusta a esa DTD.

Cada declaración está en la forma de una expresión normal para los subelementos de un elemento. Así, en la DTD de la Figura 10.7 un elemento bancario consiste en uno o más elementos cuenta, cliente o impositor; el operador | especifica “o” mientras que el operador + especifica “uno o más”. Aunque no se muestra aquí, el operador * se usa para especificar “cero o más” mientras que el operador ? se usa para especificar un elemento opcional (es decir, “cero o uno”).

El elemento `cuenta` se define para contener los subelementos `número_cuenta`, `nombre_sucursal` y `saldo` (en ese orden). De forma similar, `cliente` y `impositor` tienen los atributos en su esquema definidos como subelementos.

```
<!DOCTYPE banco [
    <!ELEMENT banco ( (cuenta|cliente|impositor)+)>
    <!ELEMENT cuenta ( número_cuenta nombre_sucursal saldo )>
    <!ELEMENT cliente ( nombre_cliente calle_cliente ciudad_cliente )>
    <!ELEMENT impositor ( nombre_cliente número_cuenta )>
    <!ELEMENT número_cuenta ( #PCDATA )>
    <!ELEMENT nombre_sucursal ( #PCDATA )>
    <!ELEMENT saldo( #PCDATA )>
    <!ELEMENT nombre_cliente( #PCDATA )>
    <!ELEMENT calle_cliente( #PCDATA )>
    <!ELEMENT ciudad_cliente( #PCDATA )>
]>
```

Figura 10.7 Ejemplo de DTD.

Finalmente, se declara a los elementos `número_cuenta`, `nombre_sucursal`, `saldo`, `nombre_cliente`, `calle_cliente` y `ciudad_cliente` del tipo #PCDATA. La palabra clave #PCDATA indica dato de texto; su nombre deriva históricamente de “parsed character data” (datos analizados de tipo carácter). Otros dos tipos especiales de declaraciones son `empty` (vacío) que indica que el elemento no tiene ningún contenido y `any` (cualquiera) que indica que no hay restricción sobre los subelementos del elemento; es decir, cualquier elemento, incluso los no mencionados en la DTD, puede ser subelemento del elemento. La ausencia de una declaración para un subelemento es equivalente a declarar explícitamente el tipo como `any`.

Los atributos permitidos para cada elemento también se declaran en la DTD. Al contrario que los subelementos no se impone ningún orden a los atributos. Los atributos se pueden especificar del tipo CDATA, ID, IDREF o IDREFS; el tipo CDATA simplemente dice que el atributo contiene datos de caracteres mientras que los otros tres no son tan sencillos; se explicarán detalladamente en breve. Por ejemplo, la siguiente línea de una DTD especifica que el elemento `cuenta` tiene un atributo del tipo `tipo_cuenta` con valor predeterminado corriente.

```
<!ATTLIST cuenta tipo_cuenta CDATA "corriente" >
```

Los atributos deben tener una declaración de tipo y una declaración predeterminada. La declaración predeterminada puede consistir en un valor predeterminado para el atributo o #REQUIRED, lo que quiere decir que se debe especificar un valor para el atributo en cada elemento, #IMPLIED, lo que significa que no se ha proporcionado ningún valor predeterminado y se puede omitir este atributo en el documento. Si un atributo tiene un valor predeterminado, para cada elemento que no tenga especificado un valor para el atributo el valor se rellena automáticamente cuando se lee el documento XML.

Un atributo de tipo ID proporciona un identificador único para el elemento; un valor de un atributo ID de un elemento no debe aparecer en ningún otro elemento del mismo documento. Sólo se permite que un único atributo de un elemento sea de tipo ID.

Un atributo del tipo IDREF es una referencia a un elemento; el atributo debe contener un valor que aparezca en el atributo ID de algún elemento en el documento. El tipo IDREFS permite una lista de referencias, separadas por espacios.

La Figura 10.8 muestra una DTD de ejemplo en la que las relaciones de la cuenta de un cliente se representan mediante los atributos ID e IDREFS, en lugar de hacerlo mediante los registros impositor. Los elementos `cuenta` usan `número_cuenta` como atributo identificador; para realizar esto se ha hecho que `número_cuenta` sea atributo de cuenta en lugar de subelemento. Los elementos `cliente` tienen un nuevo atributo identificador denominado `id_cliente`. Además, cada elemento `cliente` contiene un atributo `cuentas` del tipo IDREFS, que es una lista de identificadores de las cuentas que tiene abiertas el cliente. Cada elemento `cuenta` tiene un atributo `titulares` del tipo IDREFS, que es una lista de titulares de la cuenta.

La Figura 10.9 muestra un ejemplo de documento XML basado en la DTD de la Figura 10.8. Obsérvese que se usa un conjunto distinto de cuentas y clientes del ejemplo anterior con el fin de ilustrar mejor la característica IDREFS.

```
<!DOCTYPE banco-2 [
    <!ELEMENT cuenta ( sucursal, saldo )>
    <!ATTLIST cuenta
        número_cuenta ID #REQUIRED
        titulares IDREFS #REQUIRED >
    <!ELEMENT cliente ( nombre_cliente, calle_cliente, ciudad_cliente )>
    <!ATTLIST cliente
        id_cliente ID #REQUIRED
        cuentas IDREFS #REQUIRED >
    ... declaraciones para sucursal, saldo, nombre_cliente,
        calle_cliente y ciudad_cliente ...
]>
```

Figura 10.8 DTD con los tipos de atributo ID e IDREFS.

```

<banco-2>
    <cuenta número_cuenta="C-401" titulares="C100 C102">
        <nombre_sucursal> Centro </nombre_sucursal>
        <saldo> 500 </saldo>
    </cuenta>
    <cuenta número_cuenta="C-402" titulares="C102 C101">
        <nombre_sucursal> Navacerrada </nombre_sucursal>
        <saldo> 900 </saldo>
    </cuenta>
    <cliente id_cliente="C100" cuentas="C-401">
        <nombre_cliente>Juncal</nombre_cliente>
        <calle_cliente> Mártires </calle_cliente>
        <ciudad_cliente> Melilla </ciudad_cliente>
    </cliente>
    <cliente id_cliente="C101" cuentas="C-402">
        <nombre_cliente>Loreto</nombre_cliente>
        <calle_cliente> Montaña </calle_cliente>
        <ciudad_cliente> Cáceres </ciudad_cliente>
    </cliente>
    <cliente id_cliente="C102" cuentas="C-401 C-402">
        <nombre_cliente>María</nombre_cliente>
        <calle_cliente> Eneas </calle_cliente>
        <ciudad_cliente> Alicante </ciudad_cliente>
    </cliente>
</banco-2>

```

Figura 10.9 Datos XML con los atributos ID y IDREF.

Los atributos ID e IDREF desempeñan la misma función que los mecanismos de referencia en las bases de datos orientadas a objetos y las bases de datos relacionales orientadas a objetos permitiendo la construcción de relaciones de datos complejas.

Las definiciones de tipos de documentos están fuertemente relacionadas con la herencia del formato del documento XML. Debido a esto no son adecuadas por varios motivos para servir como estructura de tipos de XML para aplicaciones de procesamiento de datos. No obstante, un tremendo número de formatos de intercambio de datos se están definiendo en términos de DTD, puesto que fueron parte original de la norma. Veamos algunas limitaciones de las DTD como mecanismo de esquema.

- No se puede declarar el tipo de cada elemento y de cada atributo de texto. Por ejemplo, el elemento saldo no se puede restringir para que sea un número positivo. La falta de tal restricción es problemática para las aplicaciones de procesamiento e intercambio de datos, las cuales deben contener el código para verificar los tipos de los elementos y atributos.
- Es difícil usar el mecanismo DTD para especificar conjuntos desordenados de subelementos. El orden es rara vez importante para el intercambio de datos (al contrario que en el diseño de documentos, donde es crucial). Aunque la combinación de la alternativa (la operación |) y las operaciones * y + como en la Figura 10.7 permite la especificación de colecciones desordenadas de marcas, es mucho más complicado especificar que cada marca pueda aparecer solamente una vez.
- Hay una falta de tipos en ID e IDREFS. Por ello no hay forma de especificar el tipo de elemento al cual se debería referir un atributo IDREF o IDREFS. En consecuencia, la DTD de la Figura 10.8 no evita que el atributo "titulares" de un elemento cuenta se refiera a otras cuentas, aunque esto no tenga sentido.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element nombre="banco" type="TipoBanco" />
  <xs:element nombre="cuenta">
    <xs:complexType>
      <xs:sequence>
        <xs:element nombre="número_cuenta" type="xs:string"/>
        <xs:element nombre="nombre_sucursal" type="xs:string"/>
        <xs:element nombre="saldo" type="xs:decimal"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element nombre="cliente">
    <xs:complexType>
      <xs:sequence>
        <xs:element nombre="número_cliente" type="xs:string"/>
        <xs:element nombre="calle_cliente" type="xs:string"/>
        <xs:element nombre="ciudad_cliente" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element nombre="impositor">
    <xs:complexType>
      <xs:sequence>
        <xs:element nombre="nombre_cliente" type="xs:string"/>
        <xs:element nombre="número_cuenta" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType nombre="TipoBanco">
    <xs:sequence>
      <xs:element ref="cuenta" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="cliente" minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="impositor" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>

```

Figura 10.10 Versión XML Schema de la DTD de la Figura 10.7.

10.3.2 XML Schema

Un intento de reparar las deficiencias del mecanismo DTD produjo el desarrollo de un lenguaje de esquema más sofisticado, **XML Schema**. Se presenta aquí una visión general de XML Schema y se mencionan algunas áreas en las cuales mejora a las DTDs.

XML Schema define varios tipos predefinidos como *string*, *integer*, *decimal*, *date* y *boolean*. Además permite tipos definidos por el usuario, que pueden ser tipos más simples con restricciones añadidas, o tipos complejos construidos con constructores como *complexType* y *sequence*.

La Figura 10.10 muestra cómo la DTD de la Figura 10.7 se puede representar mediante XML Schema. A continuación se describen las características de XML Schema que aparecen en esa figura.

Lo primero que es preciso resaltar es que las propias definiciones en XML Schema se especifican en sintaxis XML, usando varias etiquetas definidas en XML Schema. Para evitar conflictos con las etiquetas definidas por los usuarios, se antepone a la etiqueta XML Schema con la etiqueta de espacio de nombres “*xs:*”; este prefijo se asocia con el espacio de nombres de XML Schema mediante la especificación *xmlns:xs* en el elemento raíz:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

Obsérvese que se podría usar cualquier prefijo de espacio de nombres en lugar de *xs*; por tanto se podrían reemplazar todas las apariciones de “*xs:*” por “*xsd:*” sin cambiar el significado de la definición

del esquema. A todos los tipos definidos en XML Schema se les debe anteponer este prefijo de espacio de nombres.

El primer elemento es el elemento raíz banco, cuyo tipo es TipoBanco, que se declara posteriormente. En el ejemplo se definen después los tipos de los elementos cuenta, cliente e impositor. Obsérvese que cada uno de ellos se especifica con un elemento con la etiqueta `xs:element`, cuyo cuerpo contiene la definición de tipo.

El tipo de cuenta es compuesto, y se precisa como una secuencia de elementos `número_cuenta`, `nombre_sucursal` y `saldo`. Cualquier tipo que tenga atributos o subelementos anidados se puede especificar como tipo complejo.

Alternativamente se puede especificar que el tipo de un elemento sea un tipo predefinido con el atributo `type`; obsérvese el uso de los tipos de XML Schema `xs:string` y `xs:decimal` para restringir los tipos de los elementos de datos tales como `número_cuenta`.

Finalmente, el ejemplo define el tipo `TipoBanco` para contener ninguna o más apariciones tanto de cuenta como de cliente e impositor. Obsérvese el uso de `ref` para especificar la aparición de un elemento definido anteriormente. XML Schema puede definir el número mínimo y máximo de apariciones de subelementos mediante `minOccurs` y `maxOccurs`. El valor predeterminado para las apariciones máxima y mínima es 1, por lo que se tiene que especificar explícitamente para permitir cero o más cuentas, impositores y clientes.

Los atributos se especifican usando la etiqueta `xs:attribute`. Por ejemplo, se podría haber definido `número_cuenta` como atributo añadiendo

```
<xs:attribute nombre = "número_cuenta"/>
```

dentro de la declaración del elemento `cuenta`. Al añadir el atributo `use = "required"` a la especificación anterior se declara que el atributo se debe especificar, mientras que el valor predeterminado de `use` es `optional`. Las especificaciones de atributos pueden aparecer directamente dentro de la especificación `complexType`, aunque los elementos se aniden dentro de una especificación de secuencia.

Se puede usar el elemento `xs:complexType` para crear tipos complejos con nombre; la sintaxis es la misma que la usada en el elemento `xs:complexType` de la Figura 10.10, salvo que se añade el atributo `nombre = nombreTipo` al elemento `xs:complexType`, donde `nombreTipo` es el nombre que se desea para el tipo. Se puede usar entonces el tipo nombrado para especificar el tipo de un elemento usando el atributo `type`, al igual que se han usado `xs:decimal` y `xs:string` en el ejemplo.

Además de definir tipos, un esquema relacional también permite la especificación de restricciones. XML Schema permite la especificación de claves y referencias a claves, que se corresponden con las definiciones de clave primaria y clave externa de SQL. En SQL una restricción de clave primaria o de unicidad asegura que los valores del atributo no se dupliquen en la relación. En el contexto de XML es necesario definir un ámbito en el que los valores sean únicos y formen una clave. `selector` es una expresión de ruta que define el ámbito de la restricción, y las declaraciones `field` especifican los elementos o atributos que forman la clave. Para especificar que los números de cuenta forman una clave de los elementos `cuenta` en el elemento raíz `banco` se podría añadir la siguiente especificación de restricción a la definición del esquema:

```
<xs:key nombre = "claveCuenta">
  <xs:selector xpath = "/banco/cuenta"/>
  <xs:field xpath = "número_cuenta"/>
</xs:key>
```

Se puede definir también la restricción de clave externa desde `impositor` hasta `cuenta` como sigue:

```
<xs:keyref nombre = "claveExtImpositorCuenta" refer = "claveCuenta">
  <xs:selector xpath = "/banco/impositor"/>
  <xs:field xpath = "número_cuenta"/>
</xs:key>
```

Obsérvese que el atributo `refer` especifica el nombre de la declaración de clave a la que se hace referencia, mientras que la especificación `field` identifica los atributos que hacen la referencia.

XML Schema ofrece varias ventajas con respecto a las DTDs, y actualmente se usa mucho. Entre las ventajas que se han visto en los ejemplos anteriores se encuentran las siguientes:

- Permite que el texto que aparece en los elementos esté restringido a tipos específicos tales como tipos numéricos en formatos específicos o tipos más complejos como secuencias de elementos de otros tipos.
- Permite crear tipos definidos por el usuario.
- Permite restricciones de unicidad y de clave externa.
- Está integrado con espacios de nombres para permitir a diferentes partes de un documento adaptarse a esquemas diferentes.

Además de las características estudiadas, XML Schema soporta otras características que las DTDs no soportan, tales como:

- Permite restringir tipos para crear tipos especializados, por ejemplo especificando valores máximos y mínimos.
- Permite extender los tipos complejos mediante el uso de una forma de herencia.

Esta descripción de XML Schema es sólo una visión general; para obtener más información, consultense las notas bibliográficas.

10.4 Consulta y transformación

Dado el creciente número de aplicaciones que usan XML para intercambiar, transmitir y almacenar datos, las herramientas para una gestión efectiva de datos XML están siendo cada vez más importantes. En particular las herramientas para consultar y transformar los datos XML son esenciales para extraer información de grandes cuerpos de datos XML y para convertir los datos entre distintas representaciones (esquemas) en XML. Al igual que la salida de una consulta relacional es una relación, la salida de una consulta XML puede ser un documento XML. Como resultado, la consulta y la transformación se pueden combinar en una única herramienta.

Los siguientes lenguajes proporcionan más capacidades de consulta y transformación:

- XPath es un lenguaje para expresiones de rutas de acceso y es realmente el fundamento de los otros dos lenguajes de consultas.
- XQuery es el lenguaje normalizado para la consulta de datos XML. Se ha modelado a partir de SQL, pero es significativamente diferente, ya que tiene que manejar datos XML anidados. También incorpora expresiones XPath.
- XSLT fue diseñado como lenguaje de transformación como parte del sistema de hojas de estilo XSL, que se usa para controlar el formato de los datos XML en HTML u otro lenguaje de impresión o visualización. Aunque diseñado para el formato, XSLT puede generar XML y expresar muchas consultas interesantes. Actualmente es el lenguaje más usado para la manipulación de datos XML, aunque XQuery es más adecuado para el tratamiento de bases de datos.

Se usa en todos estos lenguajes un **modelo de árbol** de datos XML. Cada documento XML se modela como un **árbol** con **nodos** correspondientes a los elementos y a los atributos. Los nodos de los elementos pueden tener nodos hijos, que pueden ser subelementos o atributos del elemento. De igual forma, cada nodo (ya sea de atributo o de elemento) distinto del elemento raíz tiene un nodo padre, que es un elemento. El orden de los elementos y de los atributos en el documento XML se modela ordenando los nodos hijos del árbol. Los términos *padre*, *hijo*, *ascendiente*, *descendiente* y *hermano* se interpretan en el modelo de árbol de datos XML.

El contenido textual de un elemento se puede modelar como un nodo de texto hijo del elemento. Los elementos que contienen texto descompuesto por subelementos interviniéntes pueden tener varios nodos de texto hijo. Por ejemplo, un elemento que contenga “Éste es un **bold** gran **/bold** libro” contiene un subelemento hijo correspondiente al elemento **bold** y dos nodos de texto hijos correspondientes a “Éste es un” y a “libro”. Puesto que dichas estructuras no se usan habitualmente en los datos de las bases de datos, se asumirá que los elementos no contienen a la vez texto y subelementos.

10.4.1 XPath

XPath trata partes de los documentos XML mediante expresiones de rutas de acceso. Se puede ver el lenguaje como una extensión a las expresiones de rutas de acceso sencillas en las bases de datos orientadas a objetos y relacionales orientadas a objetos (véase el Apartado 9.6). La versión actual de XPath es la 2.0 y nuestra descripción se basa en ella.

Cada **expresión de ruta** de XPath es una secuencia de pasos de ubicación separados por “/” (en lugar de por el operador “.” que separa los pasos en SQL:1999). El resultado de la expresión de ruta es un conjunto de nodos. Por ejemplo, en el documento de la Figura 10.9 la expresión XPath

/banco-2/cliente/nombre_cliente

devolverá estos elementos:

```
<nombre_cliente>Juncal</nombre_cliente>
<nombre_cliente>Loreto</nombre_cliente>
<nombre_cliente>María</nombre_cliente>
```

La expresión

/banco-2/cliente/nombre_cliente/text()

devolverá los mismos nombres, pero sin las etiquetas que los rodean.

Las expresiones de ruta se evalúan de izquierda a derecha. Al igual que en una jerarquía de directorios, la barra ‘/’ inicial indica la raíz del documento (obsérvese que esto es una raíz abstracta “por encima de” <banco-2>, que es la etiqueta de documento).

Al evaluar una expresión de ruta, el resultado de la ruta en cualquier punto consiste en un conjunto ordenado de nodos del documento. Inicialmente el conjunto de elementos “actual” contiene un solo nodo, la raíz abstracta. Cuando el próximo paso de una expresión de ruta es un nombre de elemento, como **cliente**, el resultado del paso consiste en los nodos correspondientes a elementos del nombre especificado que son los hijos de los elementos en el conjunto actual de elementos. Estos nodos serán el conjunto actual de elementos para el siguiente paso de la evaluación de la expresión de ruta. Los nodos devueltos en cada paso aparecen en el mismo orden que en el documento.

Ya que varios hijos pueden tener el mismo nombre, el número de nodos en el conjunto puede aumentar o disminuir en cada paso. También se puede acceder a los valores de los atributos usando el símbolo “@”. Por ejemplo, /banco-2/cuenta/@número_cuenta devuelve un conjunto con todos los valores de los atributos número_cuenta de los elementos cuenta. De forma predeterminada no se siguen los vínculos IDREF; posteriormente veremos cómo tratar con IDREF.

XPath soporta otras características:

- Los predicados de selección pueden seguir cualquier paso en una ruta y se escriben entre corchetes. Por ejemplo,

/banco-2/cuenta[saldo > 400]

devuelve los elementos cuenta con un valor saldo mayor que 400, mientras que

/banco-2/cuenta[saldo > 400]/@número_cuenta

devuelve los números de cuenta de dichas cuentas.

Se puede comprobar la existencia de un subelemento listándolo sin ningún operador de comparación; por ejemplo, si se quita simplemente “> 400”, la expresión anterior devolvería los números de cuenta de todas las cuentas que tengan un subelemento saldo, independientemente de su valor.

- XPath proporciona varias funciones que se pueden usar como parte de predicados, incluidas la comprobación de la posición del nodo actual en el orden de los hermanos y la función de agregación `count()`, que cuenta el número de nodos coincidentes con la expresión a la que se aplica. Por ejemplo, la expresión de ruta

```
/banco-2/cuenta[count(./cliente)>2]
```

devuelve las cuentas con más de dos clientes. Se pueden usar las conectivas lógicas `and` y `or` en los predicados y la función `not(...)` para la negación.

- La función `id("fu")` devuelve el nodo (si existe) con un atributo del tipo ID cuyo valor sea “fu”. La función `id` se puede incluso aplicar a conjuntos de referencias o incluso a cadenas que contengan referencias múltiples separadas por espacios vacíos, tales como IDREFS. Por ejemplo, la ruta

```
/banco-2/cuenta/id(@titular)
```

devuelve todos los clientes referenciados desde el atributo `titular` de los elementos `cuenta`.

- El operador `|` permite unir resultados de expresiones. Por ejemplo, si la DTD de `banco-2` también contiene elementos para préstamos con atributo `prestatario` del tipo IDREFS para identificar el tomador de un préstamo, la expresión

```
/banco-2/cuenta/id(@titular) | /banco-2/préstamo/id(@prestatario)
```

proporciona los clientes con cuentas o préstamos. Sin embargo, el operador `|` no se puede anidar dentro de otros operadores. También merece la pena destacar que los nodos de la unión se devuelven en el orden en el que aparecen en el documento.

- Una expresión XPath puede saltar varios niveles de nodos mediante el uso de “`//`”. Por ejemplo la expresión `/banco-2//nombre_cliente` encuentra cualquier elemento `nombre_cliente` en *cualquier lugar* por debajo del elemento `/banco-2`, independientemente del elemento en el que esté contenido y del número de niveles de inclusión presentes entre los elementos `banco-2` y `nombre_cliente`. Este ejemplo ilustra la posibilidad de buscar los datos necesarios sin un conocimiento completo del esquema.
- Cada paso en la ruta no selecciona los hijos de los nodos del conjunto actual de nodos. En realidad esto es solamente una de las distintas direcciones que puede tomar un paso en la ruta tales como padres, hermanos, ascendientes y descendientes. Aquí se omiten los detalles, pero obsérvese que “`//`”, descrito anteriormente, es una forma abreviada de especificar “todos los descendientes”, mientras que “`..`” especifica el padre.
- La función predeterminada `doc(nombre)` devuelve la raíz de un documento con nombre; el nombre puede ser un nombre de archivo o un URL. La raíz devuelta por la función se pueden usar en una expresión de ruta para acceder a los contenidos del documento. Por tanto, una expresión de ruta se puede aplicar en un documento especificado, en lugar de aplicarse al documento actual predeterminado.

Por ejemplo, si los datos bancarios de nuestro ejemplo se contuviesen en el archivo “`banco.xml`”, la siguiente expresión de ruta devolvería todas las cuentas del banco

```
doc("banco.xml")/banco/cuenta
```

La función `collection(nombre)` es similar a `doc`, pero devuelve una colección de documentos identificada por nombre.

10.4.2 XQuery

El consorcio W3C (World Wide Web Consortium) ha desarrollado XQuery como lenguaje de consulta normalizado de XML. Nuestra discusión aquí está basada en el último borrador de la norma del lenguaje disponible a principios de enero de 2005: aunque la norma final puede variar, se espera que las principales características que aquí se tratan no se modifiquen. El lenguaje XQuery procede de un lenguaje de consulta XML denominado Quilt; el propio Quilt incluía características de lenguajes anteriores, tales como XPath, estudiado en el Apartado 10.4, y otros dos lenguajes de consultas XML: XQL y XML-QL.

10.4.2.1 Expresiones FLWOR

Las consultas XQuery se basan en SQL, pero difieren significativamente de él. Se organizan en cinco secciones: **for**, **let**, **where**, **order by** y **return**. Se conocen como expresiones “FLWOR” (pronunciado “flauer”, como “flor” en inglés), cuyas letras denotan las cinco secciones.

Una expresión FLWOR sencilla que devuelve los números de cuentas de las cuentas corrientes está basada en el documento XML de la Figura 10.9, que usa ID and IDREFS:

```
for $x in /banco-2/cuenta
let $numcuenta := $x/@número_cuenta
where $x/saldo > 400
return <número_cuenta> { $numcuenta } </número_cuenta>
```

La cláusula **for** es como la cláusula **from** de SQL y proporciona una serie de variables cuyos valores son los resultados de expresiones XPath. Cuando se especifica más de una variable, los resultados incluyen el producto cartesiano de los valores posibles que las variables pueden tomar, al igual que la cláusula **from** de SQL.

La cláusula **let** simplemente permite que se asigne el resultado de expresiones XPath al nombre de las variables por simplicidad de representación. La cláusula **where**, como la cláusula **where** de SQL, ejecuta comprobaciones adicionales sobre las tuplas reunidas de la cláusula **for**. La cláusula **order by**, al igual que la cláusula **order by** de SQL, permite la ordenación de la salida. Finalmente, la cláusula **return** permite la construcción de resultados en XML.

Una consulta FLWOR no necesita tener todas las cláusulas; por ejemplo, una consulta puede contener simplemente las cláusulas **for** y **return** y omitir **let**, **where** y **order by**. La consulta XQuery anterior no contiene ninguna cláusula **order by**. De hecho, dado que esta consulta es sencilla, se puede prescindir fácilmente de la cláusula **let**, y la variable **\$numcuenta** de la cláusula **return** se podría remplazar por **\$x/@número_cuenta**. Obsérvese además que, puesto que la cláusula **for** usa expresiones XPath, se pueden producir selecciones en la expresión XPath. Por ello, una consulta equivalente puede tener solamente cláusulas **for** y **return**:

```
for $x in /banco-2/cuenta[saldo > 400]
return <número_cuenta> { $x/@número_cuenta } </número_cuenta>
```

Sin embargo, la cláusula **let** simplifica las consultas complejas. Obsérvese también que las variables asignadas por la cláusula **let** pueden contener secuencias con varios elementos o valores, si la expresión de ruta de la parte derecha los devuelve.

Obsérvese el uso de las llaves (“{}”) en la cláusula **return**. Cuando XQuery encuentra un elemento como **<número_cuenta>** que inicia una expresión, trata su contenido como texto XML normal, excepto las partes encerradas entre llaves, que se evalúan como expresiones. Así, si se hubiesen omitido las llaves en la anterior cláusula **return**, el resultado contendría varias copias de la cadena “**\$x/@número_cuenta**”, cada una de ellas encerrada en una etiqueta **número_cuenta**. Los contenidos dentro de las llaves son, no obstante, tratados como expresiones a evaluar. Este convenio se aplica incluso con las llaves entre comillas. Así, se podría modificar la consulta anterior para devolver un elemento con una etiqueta **cuenta**, con el número de cuenta como atributo, reemplazando la cláusula **return** por:

```
return <cuenta número_cuenta="{$x/@número_cuenta}" />
```

XQuery proporciona otra forma de construir elementos usando los constructores **element** y **attribute**. Por ejemplo, si la cláusula **return** de la consulta anterior se reemplaza por la siguiente cláusula **return**, la consulta devolvería elementos cuenta con número_cuenta and nombre_sucursal como atributos y saldo como subelemento.

```
return element cuenta {
    attribute número_cuenta {$x/@número_cuenta},
    attribute nombre_sucursal {$x/nombre_sucursal},
    element saldo {$x/saldo}
}
```

Obsérvese que, al igual que sucedía antes, las llaves son necesarias para tratar una cadena como una expresión a evaluar.

10.4.2.2 Reuniones

Las reuniones se especifican en XQuery de forma parecida a SQL. La reunión de los elementos impositor, cuenta y cliente de la Figura 10.1, que se escribe en XSLT en el Apartado 10.4.3, se puede escribir en XQuery de la siguiente forma:

```
for $a in /banco/cuenta,
    $c in /banco/cliente,
    $i in /banco/impositor
where $a/número_cuenta = $i/número_cuenta
    and $c/nombre_cliente = $i/nombre_cliente
return <cuenta_cliente> { $c $a } </cuenta_cliente>
```

La misma consulta se puede expresar con las selecciones especificadas como selecciones XPath:

```
for $a in /banco/cuenta,
    $c in /banco/cliente,
    $i in /banco/impositor[número_cuenta = $a/número_cuenta
        and nombre_cliente = $c/nombre_cliente]
return <cuenta_cliente> { $c $a } </cuenta_cliente>
```

Las expresiones de ruta en XQuery son las misma expresiones de ruta en XPath2.0. Las expresiones de ruta pueden devolver un único valor o elemento, o una secuencia de ellos. A falta de información de esquema puede no ser posible inferir si una expresión de ruta devuelve un único valor o una secuencia de valores. Tales expresiones pueden participar en operaciones de comparación tales como `=`, `<` y `>`.

XQuery tiene un definición interesante de las operaciones de comparación sobre secuencias. Por ejemplo, la expresión `$x/saldo > 400` tendría la interpretación usual si el resultado de `$x/saldo` fuese un único valor, pero si el resultado es una secuencia que contiene varios valores, la expresión se evalúa a cierto si al menos uno de los valores es mayor que 400. Análogamente, la expresión `$x/saldo = $y/saldo` es cierta si alguno de los valores devueltos por la primera expresión es igual a cualquier otro valor de la segunda. Si este comportamiento no es apropiado, se pueden usar las operaciones `eq`, `ne`, `lt`, `gt`, `le`, `ge` en su lugar, las cuales generan un error si cualquiera de sus entradas es una secuencia de varios valores.

10.4.2.3 Consultas anidadas

Las expresiones FLWOR de XQuery se pueden anidar en la cláusula **return** con el fin de generar anidamientos de elementos que no aparecen en el documento origen. Esta característica es similar a las subconsultas anidadas en la cláusula **from** de las consultas SQL del Apartado 9.5.3. Por ejemplo, la estructura XML mostrada en la Figura 10.4, con los elementos de la cuenta anidados en elementos cliente, se puede generar a partir de la estructura en la Figura 10.1 mediante esta consulta:

```

<banco-1> {
    for $c in /banco/cliente
    return
        <cliente>
            {$c/*}
            { for $i in /banco/impositor[nombre_cliente = $c/nombre_cliente],
                $a in /banco/cuenta[número_cuenta=$i/número_cuenta]
                return $a }
        </cliente>
} </banco-1>

```

La consulta también introduce la sintaxis `$c/*`, que se refiere a todos los hijos del nodo (o secuencia de nodos), ligados a la variable `$c`. De forma similar, `$c/text()` proporciona el contenido textual de un elemento, sin las etiquetas.

XQuery proporciona funciones de agregado tales como `sum()` y `count()` que se pueden aplicar a secuencias de elementos o valores. La función `distinct-values()` aplicada sobre una secuencia devuelve una secuencia sin duplicados. La secuencia (colección) de valores devueltos por una expresión de ruta puede tener algunos valores repetidos porque se repiten en el documento, aunque un resultado de una expresión XPath pueda contener a lo sumo una aparición de cada nodo en el documento. XQuery soporta muchas otras funciones que son comunes a XPath 2.0 y XQuery, y se pueden usar en cualquier expresión de ruta de XPath.

Para evitar conflictos, las funciones se asocian con un espacio de nombres

<http://www.w3.org/2004/10/xpath-functions>

que tiene el prefijo predeterminado `fn` para el espacio de nombres. Así, estas funciones se pueden reconocer sin ambigüedades como `fn:sum` o `fn:count`.

Aunque XQuery no proporciona un constructor **group by**, las consultas de agregación se pueden escribir usando funciones de agregado sobre expresiones de ruta o FLWOR anidadas dentro de la cláusula **return**. Por ejemplo, la siguiente consulta sobre el esquema XML `banco` calcula el saldo total de todas las cuentas de cada cliente.

```

for $c in /banco/cliente
return
    <saldo-total-cliente>
        <nombre_cliente> { $c/nombre_cliente } </nombre_cliente>
        <saldo_total> { fn:sum(
            for $i in /banco/impositor[nombre_cliente = $c/nombre_cliente],
            $a in /banco/cuenta[número_cuenta = $d/número_cuenta]
            return $a/saldo
        ) } </saldo_total>
    </saldo-total-cliente>

```

10.4.2.4 Ordenación de resultados

En XQuery los resultados se pueden ordenar si se incluye una cláusula **order by**. Por ejemplo, esta consulta tiene como salida todos los elementos `cliente` ordenados por el subelemento `nombre_cliente`:

```

for $c in /banco/cliente,
order by $c/nombre_cliente
return <cliente> { $c/* } </cliente>

```

Para ordenar de forma decreciente podemos usar **order by nombre_cliente descending**.

La ordenación se puede realizar en varios niveles de anidamiento. Por ejemplo, se puede obtener una representación anidada de la información bancaria ordenada según el nombre del cliente, con las cuentas de cada cliente ordenadas según el número de cuenta, como sigue:

```

<banco-1> {
    for $c in /banco/cliente
    order by $c/nombre_cliente
    return
        <cliente>
            { $c/*
            { for $i in /banco/impositor[nombre_cliente = $c/nombre_cliente],
                $a in /banco/cuenta[número_cuenta = $d/número_cuenta]
                order by $a/número_cuenta
                return <cuenta> { $a/* } </cuenta> }
            </cliente>
} </banco-1>

```

10.4.2.5 Funciones y tipos

XQuery proporciona una serie de funciones predefinidas, como funciones numéricas y de comparación de cadenas y funciones de manipulación. Además, XQuery soporta funciones definidas por el usuario. La siguiente función definida por el usuario devuelve una lista de todos los saldos de un cliente con un nombre especificado:

```

define function saldos(xs:string $c) as xs:decimal* {
    for $i in /banco/impositor[nombre_cliente = $c],
        $a in /banco/cuenta[número_cuenta = $d/número_cuenta]
    return $a/saldo
}

```

La especificación de tipo de los argumentos de la función y de los valores devueltos es opcional. XQuery usa el sistema de tipos de XML Schema. El prefijo `xs:` de espacio de nombres usado en el ejemplo anterior está asociado con el espacio de nombres de XML Schema de forma predefinida en XQuery.

Se puede añadir a los tipos `*` como sufijo para indicar una secuencia de valores de ese tipo; por ejemplo, la definición de la función `saldos` especifica su valor devuelto como una secuencia de valores numéricos. Los tipos se pueden especificar parcialmente; por ejemplo, el tipo `element` permite elementos con cualquier etiqueta, mientras que `element(cuenta)` permite elementos con la etiqueta `cuenta`.

XQuery realiza convierte los tipos automáticamente siempre que sea necesario. Por ejemplo, si un valor numérico representado por una cadena se compara con un tipo numérico, la conversión de tipo de cadena a número se hace automáticamente. Cuando se pasa un elemento a una función que espera una cadena, la conversión a cadena se hace concatenando todos los valores de texto contenidos (anidados) dentro del elemento. Así, la función `contains(a,b)`, que comprueba si la cadena `a` contiene a la cadena `b`, se puede usar con un elemento en su primer argumento, en cuyo caso comprueba si el elemento `a` contiene la cadena `b` anidada en cualquier lugar dentro de él. XQuery también proporciona funciones para convertir tipos. Por ejemplo, `number(x)` convierte una cadena en un número.

10.4.2.6 Otras características

XQuery ofrece una gran variedad de características adicionales tales como constructores `if-then-else`, que se pueden usar con cláusulas `return` y la cuantificación existencial y universal, que se pueden usar en predicados en cláusulas `where`. Por ejemplo, la cuantificación existencial se puede expresar en la cláusula `where` usando

`some $e in ruta satisfies P`

donde `ruta` es una expresión de ruta y `P` es un predicado que puede usar `$e`. La cuantificación universal se puede expresar usando `every` en lugar de `some`.

Como se puede ver en la descripción anterior, XQuery con XPath es un lenguaje bastante complejo, y debe manejar datos de estructura compleja. Aunque hace varios años que se definió (en un borrador),

```
<xsl:template match="/banco-2/cliente">
  <cliente>
    <xsl:value-of select="nombre_cliente"/>
  </cliente>
</xsl:template>
<xsl:template match="*"/>
```

Figura 10.11 Uso de XSLT para convertir los resultados en nuevos elementos XML.

muchas implementaciones o bien implementan un subconjunto de XQuery o bien son ineficientes en grandes conjuntos de datos.

La norma **XQJ** proporciona una interfaz de programación de aplicaciones (API) para ejecutar consultas XQuery en un sistema de bases de datos XML y para devolver los resultados XML. Su funcionalidad es similar a la API JDBC.

10.4.3 XSLT**

Una **hoja de estilo** es una representación de las opciones de formato para un documento, normalmente almacenado fuera del documento mismo, por lo que el formato está separado del contenido. Por ejemplo, una hoja de estilo para HTML puede especificar la fuente a usar en todas la cabeceras y, por ello, reemplaza un gran número de declaraciones de fuente en la página HTML. **XML XSL (Stylesheet Language)**, el lenguaje de hojas de estilo XML, estaba originalmente diseñado para generar HTML a partir de XML y es por ello una extensión lógica de hojas de estilo HTML. El lenguaje incluye un mecanismo de transformación de propósito general, denominado **XSLT (XSL Transformations**, transformaciones XSL), que se puede usar para transformar un documento XML en otro documento XML, o a otros formatos como HTML¹. Las transformaciones XSLT son bastante potentes y en realidad XSLT puede incluso actuar como lenguaje de consulta.

Las transformaciones XSLT se expresan como una serie de reglas recursivas, denominadas **plantillas**. En su forma básica las plantillas permiten la selección de nodos en un árbol XML mediante una expresión XPath. Sin embargo, las plantillas también pueden generar contenido XML nuevo de forma que esa selección y generación de contenido se pueda mezclar de formas naturales y potentes. Aunque XSLT se puede usar como lenguaje de consulta, su sintaxis y semántica son bastante distintas de las de SQL.

Una plantilla sencilla para XSLT consiste en una parte de **coincidencia** (match) y una parte de **selección** (select). Consideremos el siguiente código XSLT:

```
<xsl:template match="/banco-2/cliente">
  <xsl:value-of select="nombre_cliente"/>
</xsl:template>
<xsl:template match="*"/>
```

La instrucción `xsl:template match` contiene una expresión XPath que selecciona uno o más nodos. La primera plantilla busca coincidencias de elementos `cliente` que aparecen como hijos del elemento raíz `banco-2`. La instrucción `xsl:value-of` encerrada en la instrucción de coincidencia devuelve valores de los nodos en el resultado de la expresión XPath. La primera plantilla devuelve el valor del subelemento `nombre_cliente`; obsérvese que el valor no contiene la etiqueta de elemento.

Obsérvese que la segunda plantilla coincide con todos los nodos. Esto es necesario debido a que el comportamiento predeterminado de XSLT sobre los elementos del documento de entrada que no coinciden con ninguna plantilla es copiar sus atributos y contenidos textuales al documento de salida, y aplicar recursivamente plantillas a sus subelementos.

Cualquier texto o etiqueta de la hoja de estilo XSLT que no esté en el espacio de nombres `xsl` se copia sin cambiar en la salida. La Figura 10.11 muestra cómo usar esta característica para hacer que cada nombre de cliente del ejemplo aparezca como un subelemento del elemento “`<cliente>`”, colocando la

1. La norma XSL consiste ahora en XSLT más una norma para especificar las características de formato tales como fuentes, márgenes de página y tablas. El formato no es relevante desde el punto de vista de las bases de datos, por lo que no se tratará aquí.

```

<xsl:template match="/banco">
  <clientes>
    <xsl:apply-templates/>
  </clientes>
</xsl:template>
<xsl:template match="/cliente">
  <cliente>
    <xsl:value-of select="nombre_cliente"/>
  </cliente>
</xsl:template>
<xsl:template match="*"/>

```

Figura 10.12 Aplicación recursiva de reglas.

instrucción `xsl:value-of` entre `<cliente>` y `</cliente>`. La creación de un atributo como `id_cliente` en el elemento `cliente` generado es difícil y requiere el uso de `xsl:attribute`; véase un manual de XSLT para más detalles.

La **recursividad estructural** es una parte clave de XSLT. Hay que recordar que los elementos y subelementos naturalmente forman una estructura en árbol. La idea de la recursividad estructural es la siguiente: cuando una plantilla coincide con un elemento en la estructura de árbol XSLT puede usar la recursividad estructural para aplicar las reglas de la plantilla recursivamente a los subárboles en lugar de simplemente devolver un valor. Aplica las reglas recursivamente mediante la directiva `xsl:apply-templates`, que aparece dentro de otras plantillas.

Por ejemplo, los resultados de nuestra consulta anterior se pueden ubicar en un elemento `<clientes>` mediante la adición de una regla usando `xsl:apply-templates`, como en la Figura 10.12. La nueva regla coincide con la etiqueta externa “`banco`” y construye un documento resultado mediante la aplicación de otras plantillas a los subárboles que aparecen en el elemento `banco`, pero envolviendo los resultados en el elemento `<clientes> </clientes>` dado. Sin la recursividad forzada por la cláusula `<xsl:apply-templates/>` la plantilla devolvería `<clientes> </clientes>` y entonces aplicaría otras plantillas a los subelementos.

De hecho, la recursividad estructural es crítica para construir documentos XML bien formados, puesto que los documentos XML deben tener un único elemento de nivel superior que contenga el resto de elementos del documento.

XSLT proporciona una característica denominada **keys** (claves) que permite la búsqueda de elementos mediante el uso de valores de subelementos o atributos; los objetivos son similares a los de la función `id()` de XPath, pero las claves de XSLT permiten usar atributos distintos a los atributos `ID`. Las claves se definen mediante una directiva `xsl:key`, la cual tiene tres partes, por ejemplo:

```
<xsl:key name="numcuenta" match="cuenta" use="número_cuenta"/>
```

El atributo `nombre` se usa para distinguir claves distintas. El atributo `match` especifica los nodos a los que se aplica la clave. Finalmente, el atributo `use` especifica la expresión a usar como el valor de la clave. Obsérvese que la expresión no tiene que ser única para un elemento; esto es, más de un elemento puede tener el mismo valor de expresión. En el ejemplo la clave denominada `numcuenta` especifica que el subelemento `número_cuenta` de `cuenta` se debería usar como una clave para esa cuenta.

Las claves se pueden usar en plantillas como parte de cualquier patrón mediante la función `key`. Esta función toma el nombre de la clave y un valor y devuelve el conjunto de nodos que coinciden con ese valor. Por ello, el nodo XML para la cuenta “C-401” se puede referenciar como `key("numcuenta", "C-401")`.

Las claves se pueden usar para implementar algunos tipos de reuniones, como en la Figura 10.13. El código de la figura se puede aplicar a datos XML en el formato de la Figura 10.1. Aquí la función `key` reúne los elementos `impositor` con los elementos coincidentes `cliente` y `cuenta`. El resultado de la consulta consiste en pares de elementos `cliente` y `cuenta` encerrados entre elementos `cuenta_cliente`.

XSLT permite ordenar los nodos. Un ejemplo sencillo muestra cómo se usa `xsl:sort` en la hoja de estilo para devolver los elementos `cliente` ordenados por nombre:

```

<xsl:key name="numcuenta" match="cuenta" use="número_cuenta"/>
<xsl:key name="numcliente" match="cliente" use="nombre_cliente"/>
<xsl:template match="impositor">
  <cuenta_cliente>
    <xsl:value-of select=key("numcliente", "nombre_cliente")/>
    <xsl:value-of select=key("numcuenta", "número_cuenta")/>
  </cuenta_cliente>
</xsl:template>
<xsl:template match="*"/>

```

Figura 10.13 Reuniones en XSLT.

```

<xsl:template match="/banco">
  <xsl:apply-templates select="cliente">
    <xsl:sort select="nombre_cliente"/>
  </xsl:apply-templates>
</xsl:template>
<xsl:template match="cliente">
  <cliente>
    <xsl:value-of select="nombre_cliente"/>
    <xsl:value-of select="calle_cliente"/>
    <xsl:value-of select="ciudad_cliente"/>
  </cliente>
</xsl:template>
<xsl:template match="*"/>

```

Aquí `xsl:apply-template` tiene un atributo `select` que lo restringe para que sólo se aplique a los subelementos `cliente`. La directiva `xsl:sort` en el elemento `xsl:apply-template` hace que los nodos se ordenen *antes* de ser procesados por el siguiente conjunto de plantillas. Existen opciones para permitir la ordenación sobre varios subelementos/attributes, por valor numérico y en orden descendente.

10.5 La interfaz de programación de aplicaciones de XML

Debido a la gran aceptación de XML como una representación de datos y formato de intercambio hay gran cantidad de herramientas de software disponibles para la manipulación de datos XML. Hay dos modelos para la manipulación mediante programación de XML, cada uno disponible para su uso con una varios lenguajes de programación populares. Ambas APIs se pueden usar para analizar documentos XML y crear su representación en memoria. Se emplean para aplicaciones que manejan documentos XML individuales. Obsérvese, sin embargo, que no son adecuadas para consultar grandes colecciones de datos XML; los mecanismos de consulta declarativos tales como XPath y XQuery son mucho más adecuados para esta tarea.

Una de las APIs estándar para la manipulación de XML se basa en el *modelo de objetos documento* (Document Object Model, DOM), que trata el contenido XML como un árbol, con cada elemento representado por un nodo, denominado `DOMNode`. Los programas pueden acceder a partes del documento mediante navegación comenzando con la raíz.

Se encuentran disponibles bibliotecas DOM para la mayor parte de los lenguajes de programación más comunes y están incluso presentes en los exploradores Web, donde se pueden usar para manipular el documento mostrado al usuario. Aquí se explican algunas de las interfaces y métodos de la API DOM de Java, para mostrar cómo puede ser un DOM.

- La API DOM de Java proporciona una interfaz denominada `Node` e interfaces `Element` y `Attribute`, las cuales reciben la herencia de la interfaz `Node`.
- La interfaz `Node` proporciona métodos tales como `getParentNode()`, `getFirstChild()` y `getNextSibling()` para navegar por el árbol DOM comenzando por el nodo raíz.

- Se puede acceder a los subelementos de un elemento mediante el nombre `getElementsByTagName`(*Nombre(nombre)*), que devuelve una lista de todos los elementos hijo con un nombre de etiqueta especificado; se puede acceder a cada miembro de la lista mediante el método `item(i)`, que devuelve el *i*-ésimo elemento de la lista.
- Se puede acceder a los valores de atributo de una elemento mediante el nombre, usando el método `getAttribute(nombre)`.
- El valor de texto de un elemento se modela como nodo `Text`, que es un hijo del nodo elemento; un nodo elemento sin subelemento tiene solamente un nodo hijo. El método `getData()` del nodo `Text` devuelve el contenido de texto.

DOM también proporciona una serie de funciones para actualizar el documento mediante la adición y el borrado de hijos elemento y atributo, el establecimiento de valores de nodos, etc.

Se necesitan muchos más detalles para escribir un programa DOM real; véanse las notas bibliográficas para obtener referencias con más información.

DOM se puede utilizar para acceder a los datos XML almacenados en las bases de datos y se puede construir una base de datos XML con DOM como interfaz principal para acceder y modificar los datos. Sin embargo, la interfaz DOM no soporta ninguna forma de consulta declarativa.

La segunda interfaz de programación empleada más habitualmente, *API simple para XML* (Simple API for XML, SAX) es un modelo de *eventos* diseñado para proporcionar una interfaz común entre analizadores y aplicaciones. Esta API está construida bajo la noción de *manejadores de eventos*, que consisten en funciones especificadas por el usuario asociadas con eventos de análisis. Los eventos de análisis corresponden con el reconocimiento de partes de un documento; por ejemplo, cuando se encuentra la etiqueta de inicio de un elemento se genera un evento y cuando se encuentra su etiqueta de finalización se genera otro. Las piezas de un documento siempre se encuentran en orden desde el inicio al final.

El desarrollador de aplicaciones SAX crea funciones controladoras para cada evento y las registra. Cuando el analizador SAX lee un documento, cuando ocurre cada evento, se llama a la función controladora con parámetros que describen el evento (como la etiqueta del elemento o los contenidos textuales). Las funciones controladoras realizan entonces su función. Por ejemplo, para construir un árbol que represente los datos XML, las funciones controladoras para un evento de inicio de un atributo o elemento establecerían el nuevo elemento como el nodo donde otros nodos hijos deben adjuntarse. El elemento y evento correspondientes establecerían el padre del nodo como el nodo actual donde se pueden adjuntar más nodos hijos.

SAX requiere generalmente más esfuerzo de programación que DOM, pero ayuda a evitar la sobrecarga de la creación de un árbol DOM en situaciones donde la aplicación necesita su propia representación de datos. Si se usase DOM en tales aplicaciones, habría unas sobrecargas innecesarias de espacio y tiempo para la construcción del árbol DOM.

10.6 Almacenamiento de datos XML

Muchas aplicaciones requieren el almacenamiento de datos XML. Una forma de almacenar datos XML es como documentos en un sistema de archivos y otra es construir una base de datos de propósito especial para almacenar datos XML. Otro enfoque es convertir los datos XML a una representación relacional y almacenarla en la base de datos relacional. Hay varias alternativas para almacenar datos XML que se muestran brevemente en este apartado.

10.6.1 Almacenamiento de datos no relacionales

Existen varias alternativas para almacenar datos XML en sistemas de almacenamiento de datos no relacionales:

- **Almacenamiento en archivos planos.** Puesto que XML es principalmente un formato de archivo, un mecanismo de almacenamiento natural es simplemente un archivo plano. Este enfoque tiene muchos de los inconvenientes mostrados en el Capítulo 1 sobre el uso de sistemas de archivos como base para las aplicaciones de bases de datos. En particular hay una carencia de aislamiento

de datos, comprobaciones de integridad, atomicidad, acceso concurrente y seguridad. Sin embargo, la amplia disponibilidad de herramientas XML que funcionan sobre archivos de datos hace relativamente sencillo el acceso y consulta de datos XML almacenados en archivos. Por ello, este formato de almacenamiento puede ser suficiente para algunas aplicaciones.

- **Creación de bases de datos XML.** Las bases de datos XML son bases de datos que usan XML como modelo de datos básico. Las primeras bases de datos XML implementaban el modelo de objetos documento sobre bases de datos orientadas a objetos basadas en C++. Esto permite reusar gran parte de la infraestructura de bases de datos orientada a objetos mientras se usa una interfaz XML estándar. La adición de un XQuery o de otros lenguajes de consultas XML permite consultas declarativas. Otras implementaciones han construido toda la infraestructura de almacenamiento y de consulta XML sobre un gestor de almacenamiento que proporciona soporte transaccional.

Aunque se han diseñado varios sistemas de bases de datos específicamente para almacenar datos XML, el desarrollo de un sistema gestor de bases de datos desde cero es una tarea muy compleja. Se debería dar soporte no sólo al almacenamiento y consulta de datos XML, sino también a otras características tales como transacciones, seguridad, soporte de acceso a datos desde clientes y capacidades de administración. Parece entonces razonable usar un sistema de bases de datos existente para proporcionar estas características e implementar el almacenamiento y la consulta XML, bien sobre la abstracción relacional, bien como una capa paralela a ella. Estas alternativas se estudian en el Apartado 10.6.2.

10.6.2 Bases de datos relacionales

Puesto que las bases de datos relacionales se usan ampliamente en aplicaciones existentes, es una gran ventaja almacenar datos XML en bases de datos relacionales de forma que se pueda acceder a los datos desde aplicaciones existentes.

La conversión de datos XML a una forma relacional es normalmente muy sencilla si los datos se han generado en un principio desde un esquema relacional y XML se usó simplemente como un formato de intercambio de datos para datos relacionales. Sin embargo, hay muchas aplicaciones donde los datos XML no se han generado desde un esquema relacional y la traducción de éstos a una forma relacional de almacenamiento puede no ser tan sencilla. En particular los elementos anidados y los elementos que se repiten (correspondientes a atributos con valores de conjunto) complican el almacenamiento de los datos XML en un formato relacional. Se encuentran disponibles diversas alternativas que se describen a continuación.

10.6.2.1 Almacenamiento con cadenas

Los pequeños documentos XML se pueden almacenar como cadenas (**clob**) en tuplas de una base de datos relacional. Los documentos XML grandes cuyo elemento de nivel superior tenga muchos hijos se puede tratar almacenando cada elemento hijo como una cadena en una tupla separada de la base de datos. Por ejemplo, los datos XML de la Figura 10.1 se podrían almacenar como un conjunto de tuplas en una relación *elementos(datos)*, donde el atributo *datos* de cada tupla almacena un elemento XML (cuenta, cliente o impositor) en forma de cadena.

Aunque esta representación es fácil de usar, el sistema de la base de datos no conoce el esquema de los elementos almacenados. Como resultado no es posible consultar los datos directamente. En realidad no es siquiera posible implementar selecciones sencillas tales como buscar todos los elementos cuenta o encontrar el elemento cuenta cuyo número de cuenta sea C-401, sin explorar todas las tuplas de la relación y examinar los contenidos de la cadena.

Una solución parcial a este problema es almacenar distintos tipos de elementos en relaciones diferentes y también almacenar los valores de algunos elementos críticos como atributos de la relación que permite la indexación. Así, en nuestro ejemplo, las relaciones serían *elementos_cuenta*, *elementos_cliente* y *elementos_impositor*, cada una con un atributo *datos*. Cada relación puede tener atributos extra para almacenar los valores de algunos subelementos tales como *número_cuenta* o *nombre_cliente*. Por ello se puede responder eficientemente con esta representación a una consulta que requiera los elementos cuenta con un número de cuenta especificado. Tal enfoque depende del tipo de información sobre los datos XML, tales como la DTD de los datos.

Algunos sistemas de bases de datos, tales como Oracle, soportan **índices de función**, que pueden ayudar a evitar la duplicación de atributos entre la cadena XML y los atributos de la relación. A diferencia de los índices normales, que se construyen sobre los valores de atributos, los índices de función se pueden construir sobre el resultado de aplicar funciones definidas por el usuario a las tuplas. Por ejemplo, se puede construir un índice de función sobre una función definida por el usuario que devuelve el valor del subelemento *número_cuenta* de la cadena XML en una tupla. El índice se puede entonces usar de la misma forma que un índice sobre un atributo *número_cuenta*.

Estos enfoques tienen el inconveniente de que una gran parte de la información XML se almacena en cadenas. Es posible almacenar toda la información en relaciones en alguna de las formas que se examinan a continuación.

10.6.2.2 Representación con árboles

Cualquier dato XML se puede modelar como árbol y almacenar mediante el uso de un par de relaciones:

$$\begin{aligned} & \text{nodos}(id, tipo, etiqueta, valor) \\ & \text{hijo}(id_hijo, id_padre) \end{aligned}$$

A cada elemento y atributo de los datos XML se le proporciona un identificador único. Una tupla insertada en la relación *nodos* para cada elemento y atributo con su identificador (*id*), su tipo (atributo o elemento), el nombre del elemento o atributo (*etiqueta*) y el valor textual del elemento o atributo (*valor*). La relación *hijo* se usa para guardar el elemento padre de cada elemento y atributo. Si la información de orden de los elementos y atributos se debe preservar, se puede agregar un atributo adicional, *posición*, a la relación *hijo* para indicar su posición relativa entre los hijos del padre. Como ejercicio se pueden representar los datos XML de la Figura 10.1 mediante el uso de esta técnica.

Esta representación tiene la ventaja de que toda la información XML se puede representar directamente de forma relacional y se pueden trasladar muchas consultas XML a consultas relacionales y ejecutar dentro del sistema de la base de datos. Sin embargo, tiene el inconveniente de que cada elemento se divide en muchas piezas y se necesita gran número de combinaciones para volver a ensamblar los subelementos en un solo elemento.

10.6.2.3 Representación con relaciones

Según este enfoque, los elementos XML cuyo esquema es conocido se representan mediante relaciones y atributos. Los elementos cuyo esquema es desconocido se almacenan como cadenas o como una representación en árbol.

Se crea una relación para cada tipo de elemento (incluyendo subelementos) cuyo esquema sea conocido y cuyo tipo sea complejo (es decir, que contenga atributos o subelementos). Los atributos de la relación se definen como sigue:

- Todos los atributos de estos elementos se almacenan como atributos de tipo cadena de la relación.
- Si un subelemento del elemento es de tipo simple (es decir, no contiene atributos o subelementos), se añade un atributo a la relación para representarlo. El tipo predeterminado del atributo es cadena, pero si el subelemento tuviese el tipo XML Schema se podría usar el tipo SQL correspondiente.

Por ejemplo, el subelemento *número_cuenta* del elemento *cuenta* se convierte en atributo de la relación *cuenta*.

- En otro caso, se crea una relación correspondiente al subelemento (usando recursivamente las mismas reglas en sus subelementos). Además:
 - Se añade un atributo identificador a las relaciones que representen al elemento (el atributo identificador se añade sólo una vez aunque el elemento tenga varios subelementos).
 - Se añade el atributo *id_padre* a la relación que representa al subelemento, almacenando el identificador de su elemento padre.
 - Si es necesario conservar el orden, se añade el atributo *posición* a la relación que representa al subelemento.

Obsérvese que cuando se aplica este enfoque a los elementos bajo el elemento raíz de la DTD de los datos de la Figura 10.1 se vuelve al esquema relacional original que se ha usado en capítulos anteriores.

Este enfoque admite diferentes variantes. Por ejemplo, las relaciones correspondientes a subelementos que pueden aparecer una vez o sumo se pueden “aplanar” en la relación padre trasladando todos los atributos a ella. Las notas bibliográficas proporcionan referencias a enfoques diferentes para representar datos XML como relaciones.

10.6.2.4 Publicación y fragmentación de datos XML

Cuando se usa XML para intercambiar datos entre aplicaciones de negocio, los datos se originan muy frecuentemente en bases de datos relacionales. Los datos en las bases de datos relacionales deben ser *publicados*, esto es, convertidos a formato XML para su exportación a otras aplicaciones. Los datos de entrada deben ser *fragmentados*, es decir, convertidos de XML a un formato normalizado de relación y almacenado en una base de datos relacional. Aunque el código de la aplicación pueda ejecutar las operaciones de publicación y fragmentación, las operaciones son tan comunes que la conversión se debería realizar de forma automática, sin escribir ningún código en la aplicación, siempre que sea posible. Por ello, los fabricantes de bases de datos están trabajando para dar *capacidades XML* a sus productos de bases de datos.

Una base de datos con capacidades XML permite una correspondencia automática de su modelo relacional interno con XML. Esta correspondencia puede ser sencilla o compleja. Una correspondencia sencilla podría asignar un elemento XML a cada fila de una tabla y hacer de cada columna un subelemento del elemento XML. El esquema XML de la Figura 10.1 se puede crear de una representación relacional de información bancaria usando dicha correspondencia. Esta correspondencia es sencilla de generar automáticamente. Esta vista XML de los datos relacionales se pueden tratar como un documento XML *virtual*, y las consultas XML se pueden ejecutar en el documento XML *virtual*. Una correspondencia más complicada permitiría que se crearan estructuras anidadas. Las extensiones de SQL con consultas anidadas en la cláusula **select** se han desarrollado para permitir una creación fácil de salida XML anidado. Estas extensiones se describen en el Apartado 10.6.3.

También se han definido correspondencias para fragmentar datos XML en una representación relacional. Para los datos XML creados de una representación relacional la correspondencia requerida para fragmentar los datos es sencilla e inversa a la correspondencia usada para publicar los datos. Para el caso general se puede generar la correspondencia como se describe en el Apartado 10.6.2.3.

10.6.2.5 Almacenamiento nativo en bases de datos relacionales

Recientemente, las bases de datos han comenzado a dar soporte al **almacenamiento nativo** de XML. Estos sistemas almacenan datos XML como cadenas o en representaciones binarias más eficientes, sin convertir los datos a la forma relacional. Se introduce el nuevo tipo de datos **xml** para representar datos XML, aunque los tipos CLOB y BLOB pueden proporcionar el mecanismo subyacente de almacenamiento. Se incluyen lenguajes de consultas XML tales como XPath y XQuery para las consultas de datos XML.

Se puede utilizar una relación con un atributo de tipo **xml** para almacenar una colección de documentos XML; cada documento se almacena como un valor de tipo **xml** en una tupla diferente. Se crean índices ad hoc para indexar los datos XML.

Varios sistemas de bases de datos proporcionan soporte nativo para datos XML. Proporcionan un tipo de datos **xml** y permiten que las consultas XQuery se incorporen dentro de las consultas SQL. Cada consulta XQuery se puede ejecutar en un solo documento XML y se puede incorporar dentro de una consulta SQL para permitir que se ejecute en cada colección de documentos, con cada documento almacenado en una tupla diferente. Por ejemplo, véase el Apartado 29.11 para más detalles sobre el soporte nativo de XML en SQL Server 2005 de Microsoft.

10.6.3 SQL/XML

La norma SQL/XML recientemente publicada define una extensión normalizada de SQL, que permite la creación de respuesta XML anidada. La norma contiene varias partes, incluyendo una manera estándar

de identificar los tipos SQL con los tipos de XML Schema, y una manera estándar de identificar esquemas relacionales con esquemas XML, así como extensiones del lenguaje de consultas SQL.

Por ejemplo, la representación SQL/XML de la relación *cuenta* tendría un esquema XML con *cuenta* como elemento más externo, con cada tupla asociada a un elemento *fila* de XML, y cada atributo de la relación asociada a un elemento de XML del mismo nombre (con algunos convenios para resolver incompatibilidades con los caracteres especiales de los nombres). También se puede asociar un esquema SQL completo, con varias relaciones, a XML de manera parecida. La Figura 10.14 muestra la representación SQL/XML del esquema *banco* que contiene las relaciones *cuenta*, *cliente* e *impositor*.

SQL/XML añade varios operadores y operaciones de agregación a SQL para permitir la construcción de la salida XML directamente a partir de SQL extendido. La función **xmlelement** se puede utilizar para crear elementos de XML, mientras que se puede usar **xmlattributes** para crear atributos, según se ilustra en la siguiente consulta.

```
select xmlelement( name "cuenta",
                   xmlattributes( número_cuenta as número_cuenta),
                   xmlelement( name "nombre_sucursal", nombre_sucursal),
                   xmlelement( name "saldo", saldo))
  from cuenta
```

Esta consulta crea un elemento XML para cada cuenta, con el número de cuenta representado como atributo, y nombre de la sucursal y saldo como subelementos. El resultado sería como los elementos cuenta mostrados en la Figura 10.9, sin el atributo tenedores. El operador **xmlattributes** crea el nombre de atributo XML usando el nombre del atributo SQL, que se puede cambiar usando la cláusula **as** según se ha visto.

El operador **xmlforest** simplifica la construcción de las estructuras XML. Su sintaxis y comportamiento son similares a los de **xmlattributes**, excepto que crea un bosque (colección) de subelementos, en vez de una lista de atributos. Toma varios argumentos, creando un elemento para cada argumento con el nombre SQL del atributo usado como nombre del elemento XML. El operador del **xmlconcat** se puede utilizar para concatenar los elementos creados por subexpresiones en un bosque.

Cuando el valor SQL usado para construir un atributo es nulo, se omite. Los valores nulos se omiten cuando se construye el cuerpo de un elemento.

SQL/XML también proporciona la nueva función de agregación **xmlagg** que crea un bosque (colección) de elementos XML a partir de la colección de los valores a los que se aplica. La consulta siguiente crea un elemento para cada sucursal, contenido como subelementos todos los números de cuenta en esa sucursal. Puesto que la pregunta tiene una cláusula **group by nombre_sucursal**, la función de agregación se aplica a todas las cuentas de cada sucursal, lo que crea una secuencia de elementos número de cuenta.

```
select xmlelement( name "sucursal",
                   nombre_sucursal,
                   xmlagg ( xmlforest(número_cuenta)
                           order by número_cuenta))
  from cuenta
 group by nombre_sucursal
```

SQL/XML permite que la secuencia creada por **xmlagg** esté ordenada, según se ilustra en la consulta anterior. Véanse las notas bibliográficas para consultar referencias sobre más información de SQL/XML.

10.7 Aplicaciones XML

A continuación se describen varias aplicaciones de XML para el almacenamiento y comunicación (intercambio) de datos para el acceso a servicios Web (recursos de información).

```

<banco>
  <cuenta>
    <fila>
      <número_cuenta> C-101 </número_cuenta>
      <nombre_sucursal> Centro </nombre_sucursal>
      <saldo> 500 </saldo>
    </fila>
    <fila>
      <número_cuenta> C-102 </número_cuenta>
      <nombre_sucursal> Navacerrada </nombre_sucursal>
      <saldo> 400 </saldo>
    </fila>
    <fila>
      <número_cuenta> C-201 </número_cuenta>
      <nombre_sucursal> Galapagar </nombre_sucursal>
      <saldo> 900 </saldo>
    </fila>
  </cuenta>
  <cliente>
    <fila>
      <nombre_cliente> González </nombre_cliente>
      <calle_cliente> Arenal </calle_cliente>
      <ciudad_cliente> La Granja </ciudad_cliente>
    </fila>
    <fila>
      <nombre_cliente> López </nombre_cliente>
      <calle_cliente> Mayor </calle_cliente>
      <ciudad_cliente> Peguerinos </ciudad_cliente>
    </fila>
  </cliente>
  <impositor>
    <fila>
      <número_cuenta> C-101 </número_cuenta>
      <nombre_cliente> González </nombre_cliente>
    </fila>
    <fila>
      <número_cuenta> C-201 </número_cuenta>
      <nombre_cliente> González </nombre_cliente>
    </fila>
    <fila>
      <número_cuenta> C-102 </número_cuenta>
      <nombre_cliente> López </nombre_cliente>
    </fila>
  </impositor>
</banco>

```

Figura 10.14 Representación SQL/XML de información bancaria.

10.7.1 Almacenamiento de datos con estructura compleja

Muchas aplicaciones necesitan almacenar datos que están estructurados, pero no se modelan fácilmente como relaciones. Considérense, por ejemplo, las preferencias de usuario que una aplicación como un navegador debe almacenar. Normalmente existe un gran número de campos, como la página de inicio, ajustes de seguridad, de idioma y de la representación que se deben registrar. Algunos de los campos son multivalorados, por ejemplo, una lista de sitios de confianza o quizás listas ordenadas, por ejem-

plo, una lista de marcadores. Las aplicaciones han usado tradicionalmente algún tipo de representación textual para almacenar estos datos. Hoy, un gran número de estas aplicaciones prefieren almacenar esta información de configuración en formato XML. Las representaciones textuales ad hoc usadas anteriormente requieren el esfuerzo de diseñar y crear los programas de análisis para leer el archivo y convertir los datos en una forma que el programa pueda utilizar. La representación XML evita estos dos pasos.

Las representaciones basadas en XML se han propuesto incluso como normas para almacenar los documentos, los datos de hojas de cálculo y otros datos que son parte de paquetes de aplicaciones de oficina.

XML también se usa para representar datos de estructura compleja que se deben intercambiar entre diversas partes de una aplicación. Por ejemplo, un sistema de bases de datos puede representar planes de ejecución de consultas (una expresión del álgebra relacional con información adicional sobre la forma en que se deben ejecutar las operaciones) usando XML. Esto permite que una parte del sistema genere el plan de ejecución de la consulta y otra parte para mostrarla sin usar una estructura de datos compartida. Por ejemplo, los datos se pueden generar en un sistema servidor y enviarse a un sistema cliente donde se muestren.

10.7.2 Formatos normalizados para el intercambio de datos

Se han desarrollando normas basadas en XML para la representación de datos para una gran variedad de aplicaciones especializadas que van desde aplicaciones de negocios tales como banca y transportes a aplicaciones científicas tales como química y biología molecular. Veamos algunos ejemplos:

- La industria química necesita información sobre los compuestos químicos tales como su estructura molecular y una serie de propiedades importantes tales como los puntos de fusión y ebullición, valores caloríficos y solubilidad en distintos disolventes. *ChemML* es una norma para representar dicha información.
- En el transporte, los transportes de mercancías y los agentes de aduana e inspectores necesitan los registros de los envíos que contengan información detallada sobre los bienes que están siendo transportados, por quién y desde dónde se han enviado, a quién y a dónde se envían, el valor monetario de los bienes, etc.
- Un mercado en línea en que los negocios pueden vender y comprar bienes (el llamado mercado B2B—business-to-business, entre negocios) requiere información tal como los catálogos de producto, incluyendo descripciones detalladas de los productos e información de los precios, inventarios de los productos, cuotas para una venta propuesta y pedidos de compras. Por ejemplo las normas *RosettaNet* para aplicaciones de negocios electrónicos definen los esquemas XML y la semántica para la representación de datos, así como normas para el intercambio de mensajes.

El uso de esquemas relacionales normalizados para modelar requisitos de datos tan complejos resultaría en un gran número de relaciones que no se corresponden directamente con los objetos que se modelan. Las relaciones frecuentemente tienen un gran número de atributos; la representación explícita de nombres de atributos y elementos con sus valores en XML ayuda a evitar confusiones entre los atributos. Las representaciones de elementos anidados ayudan a reducir el número de relaciones que se deben representar así como el número de combinaciones requeridas para obtener la información, con el posible coste de redundancia. Por ejemplo, en nuestro ejemplo bancario el listado de clientes con elementos cuenta anidados en elementos cuenta, como en la Figura 10.4, resulta en un formato más adecuado para algunas aplicaciones—en particular para su legibilidad—que la representación normalizada de la Figura 10.1.

10.7.3 Servicios Web

Las aplicaciones requieren a menudo datos externos o de otro departamento de la empresa en una base de datos diferente. En estas situaciones, la empresa externa o el departamento no están dispuestos a permitir acceso directo a su base de datos usando SQL, sino a proporcionar información limitada a través de interfaces predefinidas.

Cuando son las personas quienes deben usar directamente la información, las empresas proporcionan formularios Web donde los usuarios pueden introducir valores y conseguir la información deseada en HTML. Sin embargo, hay muchas aplicaciones en las que son programas los que acceden a esta información, en lugar de personas. Proporcionar los resultados de una consulta en XML es un requisito claro. Además, tiene sentido especificar los valores de la entrada a la consulta también en formato de XML.

En efecto, el proveedor de información define los procedimientos cuya entrada y salida están en formato XML. El protocolo HTTP se utiliza para comunicar la información de entrada y salida, puesto que se usa en gran medida y puede pasar a través de los cortafuegos que las instituciones emplean para mantener aislado el tráfico indeseado de Internet.

El **protocolo simple de acceso a objetos (SOAP, Simple Object Access Protocol)** define una norma para invocar procedimientos usando XML para representar la entrada y salida de los procedimientos. SOAP define un esquema XML estándar para representar el nombre del procedimiento y de los indicadores de estado del resultado, como fallo y error. Los parámetros y resultados de los procedimientos son datos XML dependientes de las aplicaciones incorporados en las cabeceras XML de SOAP.

HTTP se usa normalmente como el protocolo de transporte para SOAP, pero también se puede emplear un protocolo basado en mensajes (como correo electrónico sobre el protocolo SMTP). Actualmente la norma SOAP se utiliza mucho. Por ejemplo, Amazon y Google proporcionan procedimientos basados en SOAP para realizar búsquedas y otras actividades. Otras aplicaciones que proporcionen servicios de alto nivel a los usuarios pueden invocar a estos procedimientos. La norma SOAP es independiente del lenguaje de programación subyacente, y es posible que un sitio que funciona con un lenguaje, como C#, invoque un servicio que funcione en otro lenguaje, como Java.

Un sitio que proporcione tal colección de procedimientos SOAP se denomina **servicio Web**. Se han definido varias normas para dar soporte a los servicios Web. El **lenguaje de descripción de servicios Web (WSDL, Web Services Descripción Language)** es un lenguaje usado para describir las capacidades de los servicios Web. WSDL proporciona las capacidades que la definición de la interfaz (o las definiciones de las funciones) proporciona en un lenguaje de programación tradicional, especificando las funciones disponibles y sus tipos de entrada y de salida. Además, WSDL permite la especificación del URL y del número de puerto de red que se utilizarán para invocar el servicio Web. Hay también una norma denominada **descripción, descubrimiento e integración universales (UDDI, Universal Descripción, Discovery and Integration)**, que define la forma en que se puede crear un directorio de los servicios Web disponibles y cómo un programa puede buscar en el directorio para encontrar un servicio Web que satisfaga sus necesidades.

El siguiente ejemplo ilustra el valor de los servicios Web. Una compañía de líneas aéreas puede definir un servicio Web que proporcione el conjunto de procedimientos que un sitio Web de una agencia de viajes puede invocar; pueden incluir procedimientos para encontrar horarios de vuelo y la información de tasas, así como para hacer reservas. El sitio Web de la agencia puede interactuar con varios servicios Web proporcionados por diversas líneas aéreas, hoteles y otras compañías, proporcionar la información del viaje a los clientes y hacer reservas. Al dar soporte a los servicios Web, las empresas permiten que se construya un servicio útil que integre servicios individuales. Los usuarios pueden interactuar con un solo sitio Web para hacer sus reservas sin tener que entrar en contacto con varios sitios Web diferentes.

Para llamar a un servicio Web los clientes deben preparar un mensaje XML apropiado bajo SOAP y enviarlo al servicio; cuando consigue el resultado en XML, el cliente debe extraer entonces la información del resultado XML. Existen APIs estándar en lenguajes como Java y C# para crear y extraer la información de los mensajes SOAP.

Véanse las notas bibliográficas para referencias a más información sobre los servicios Web.

10.7.4 Mediación de datos

La compra comparada es un ejemplo de aplicación de mediación de datos en la que los datos sobre elementos, inventario, precio y costes de envío se extraen de una serie de sitios Web que ofrecen un elemento concreto de venta. La información agregada resultante es significativamente más valiosa que la información individual ofrecida por un único sitio.

Un gestor financiero personal es una aplicación similar en el contexto de la banca. Consideremos un consumidor con una gran cantidad de cuentas a gestionar, tales como cuentas bancarias, cuentas de aho-

rro y cuentas de jubilación. Supóngase que estas cuentas pueden estar en distintas instituciones. Es un reto importante proporcionar una gestión centralizada de todas las cuentas de un cliente. La mediación basada en XML soluciona el problema extrayendo una representación XML de la información de la cuenta desde los sitios Web respectivos de las instituciones financieras donde está cada cuenta. Esta información se puede extraer fácilmente si la institución la exporta a un formato XML estándar, por ejemplo, como servicio Web. Para aquellas que no lo hacen se usa un software *envolvente* para generar datos XML a partir de las páginas Web HTML devueltas por el sitio Web. Las aplicaciones envolventes necesitan un mantenimiento constante puesto que dependen de los detalles de formato de las páginas Web, que cambian constantemente. No obstante, el valor proporcionado por la mediación frecuentemente justifica el esfuerzo requerido para desarrollar y mantener las aplicaciones envolventes.

Una vez que las herramientas básicas están disponibles para extraer la información de cada fuente, se usa una aplicación *mediadora* para combinar la información extraída bajo un único esquema. Esto puede requerir más transformación de los datos XML de cada sitio, puesto que los distintos sitios pueden estructurar la misma información de una forma diferente. Por ejemplo, uno de los bancos puede exportar información en el formato de la Figura 10.1 aunque otros pueden usar la formato anidado de la Figura 10.4. También pueden usar nombres diferentes para la misma información (por ejemplo `num_cuenta` e `id_cuenta`), o pueden incluso usar el mismo nombre para información distinta. El mediador debe decidir sobre un único esquema que representa toda la información requerida, y debe proporcionar código para transformar los datos entre diferentes representaciones. Dichos temas se estudiarán con más detalle en el Apartado 22.8 en el contexto de bases de datos distribuidas. Los lenguajes de consultas XML tales como XSLT y XQuery desempeñan un papel importante en la tarea de transformación entre distintas representaciones XML.

10.8 Resumen

- Al igual que el lenguaje de marcas de hipertexto HTML (Hyper-Text Markup Language), en que está basado la Web, el lenguaje de marcas extensible, XML (Extensible Markup Language), es descendiente del lenguaje estándar generalizado de marcas (SGML, Standard Generalized Markup Language). XML estaba pensado inicialmente para proporcionar marcas funcionales para documentos Web, pero se ha convertido de facto en el formato de datos estándar para el intercambio de datos entre aplicaciones.
- Los documentos XML contienen elementos con etiquetas de inicio y finalización correspondientes que indican el comienzo y finalización de un elemento. Los elementos puede tener subelementos anidados a ellos, a cualquier nivel de anidamiento. Los elementos pueden también tener atributos. La elección entre representar información como atributos o como subelementos suele ser arbitraria en el contexto de la representación de datos.
- Los elementos pueden tener un atributo del tipo `ID` que almacene un identificador único para el elemento. Los elementos también pueden almacenar referencias a otros elementos mediante el uso de atributos del tipo `IDREF`. Los atributos del tipo `IDREFS` pueden almacenar una lista de referencias.
- Los documentos pueden opcionalmente tener su esquema especificado mediante una definición de tipos de documentos (DTD, Document Type Declaration). La DTD de un documento especifica los elementos que pueden aparecer, cómo se pueden anidar y los atributos que puede tener cada elemento. Aunque las DTDs se usan mucho, tienen graves limitaciones. Por ejemplo, no proporcionan un sistema de tipos.
- XML Schema es ahora el mecanismo estándar para especificar el esquema de los documentos XML. Proporciona un gran conjunto de tipos básicos, así como constructores para crear tipos complejos y para especificar restricciones de integridad, incluidas las claves y las claves externas (`keyref`).

- Los datos XML se pueden representar como estructuras en árbol, como nodos correspondientes a los elementos y atributos. El anidamiento de elementos se refleja mediante la estructura padre-hijo de la representación en árbol.
- Las expresiones de ruta se pueden usar para recorrer la estructura de árbol XML y así localizar los datos requeridos. XPath es un lenguaje normalizado para las expresiones de rutas de acceso y permite especificar los elementos requeridos mediante una ruta parecida a un sistema de archivos y además permite la selección y otras características. XPath también forma parte de otros lenguajes de consultas XML.
- El lenguaje XQuery es el estándar para la consulta de datos XML. Tiene una estructura similar a la de SQL, con cláusulas **for**, **let**, **where**, **order by** y **return**. Sin embargo, soporta muchas extensiones para tratar con la naturaleza en árbol de XML y para permitir la transformación de documentos XML en otros documentos con una estructura significativamente diferente. Las expresiones de ruta XPath forman parte de XQuery. XQuery soporta consultas anidadas y funciones definidas por el usuario.
- El lenguaje XSLT se diseñó originalmente como el lenguaje de transformación para una aplicación de hojas de estilo, en otras palabras, para aplicar información de formato a documentos XML. Sin embargo, XSLT ofrece características bastante potentes de consulta y transformación y está ampliamente disponible, por lo que se usa para consultar datos XML.
- Las APIs DOM y SAX se usan mucho para el acceso mediante programación a datos XML. Estas APIs están disponibles para varios lenguajes de programación.
- Los datos XML se pueden almacenar de varias formas distintas. Los datos XML se pueden almacenar en sistemas de archivos, o en bases de datos XML, que emplean XML como representación interna.

XML se puede almacenar como cadenas en una base de datos relacional. Alternativamente, las relaciones pueden representar datos XML como árboles. Otra alternativa es que los datos XML se pueden hacer corresponder con relaciones de la misma forma que se hacen corresponder los esquemas E-R con esquemas relacionales. El almacenamiento nativo de XML en las bases de datos relacionales se facilita añadiendo el tipo de datos **xml** a SQL.

- XML se utiliza en gran variedad de aplicaciones, como el almacenamiento de datos complejos, el intercambio de datos entre empresas de forma estándar, la mediación de datos y los servicios Web. Los servicios Web proporcionan una interfaz de llamada a procedimientos remotos, con XML como mecanismo para codificar los parámetros y los resultados.

Términos de repaso

- Lenguaje de marcas extensible (Extensible Markup Language, XML).
- Lenguaje de marcas de hipertexto (HyperText Markup Language, HTML).
- Lenguaje estándar generalizado de marcas (Standard Generalized Markup Language, SGML).
- Lenguaje de marcas.
- Marcas.
- Autodocumentado.
- Elemento.
- Elemento raíz.
- Elementos anidados.
- Atributos.
- Espacio de nombres.
- Espacio de nombres predeterminado.
- Definición del esquema.
- Definición de tipos de documentos (Document Type Definition, DTD).
 - ID.
 - IDREF e IDREFS.
- XML Schema.
 - Tipos simples y complejos.
 - Tipo secuencia.
 - Clave y keyref.
 - Restricciones de aparición.
- Modelo de árbol de datos XML.
- Nodos.

- Consulta y transformación.
- Expresiones de ruta.
- XPath.
- XQuery.
 - Expresiones FLWOR:
 - **for**.
 - **let**.
 - **order by**.
 - **where**.
 - **return**.
 - Reuniones.
 - Expresión FLWOR anidada.
 - Ordenación.
- XSL (XML Stylesheet Language), lenguaje de hojas de estilo XML.
- Transformaciones XSL (XSL Transformations, XSLT).
 - Plantillas:
 - **match** (coincidencia).
 - **select** (selección).
 - Recursividad estructural.
 - Claves.
 - Ordenación.
- API XML.
- Modelo de objetos documento (Document Object Model, DOM).
- API simple para XML (Simple API for XML, SAX).
- Almacenamiento de datos XML.
 - En almacenamientos de datos no relacionales.
 - En bases de datos relacionales.
 - Almacenamiento como cadena.
 - Representación en árbol.
 - Representación con relaciones.
 - Publicación y fragmentación.
 - Base de datos con capacidades XML.
 - Almacenamiento nativo.
 - SQL/XML.
- Aplicaciones XML.
 - Almacenamiento de datos complejos.
 - Intercambio de datos.
 - Mediación de datos.
 - SOAP.
 - Servicios Web.

Ejercicios prácticos

- 10.1 Dese una representación alternativa de la información bancaria que contenga los mismos datos que en la Figura 10.1, pero usando atributos en lugar de subelementos. Dese también la DTD para esta representación.
- 10.2 Dese la DTD para una representación XML del siguiente esquema relacional anidado:
- ```

$$\begin{aligned} Emp &= (\text{nombre}, \text{ConjuntoHijos } \textbf{setof}(Hijos), \text{ConjuntoMaterias } \textbf{setof}(Materias)) \\ Hijos &= (\text{nombre}, \text{Cumpleaños}) \\ Cumpleaños &= (\text{día}, \text{mes}, \text{año}) \\ Materias &= (\text{tipo}, \text{ConjuntoExámenes } \textbf{setof}(Exámenes)) \\ Exámenes &= (\text{año}, \text{ciudad}) \end{aligned}$$


```
- 10.3 Escríbase una consulta en XPath sobre la DTD del Ejercicio 10.2 para listar todos los tipos de materia de *Emp*.
- 10.4 Escríbase una consulta en XQuery en la representación XML de la Figura 10.1 para encontrar el saldo total de cada sucursal teniendo en cuenta todas las cuentas.
- 10.5 Escríbase una consulta en XQuery en la representación XML de la Figura 10.1 para calcular la reunión externa por la izquierda de los elementos **impositor** con los elementos **cuenta**. *Sugerencia:* se puede usar la cuantificación universal.
- 10.6 Escríbanse consultas en XQuery y XSLT que devuelvan los elementos **cliente** con los elementos **cuenta** asociados anidados en los elementos **cliente**, dada la representación de la información bancaria usando ID e IDREFS de la Figura 10.9.
- 10.7 Dese un esquema relacional para representar la información bibliográfica como se especifica en el fragmento DTD de la Figura 10.15. El esquema relacional debe registrar el orden de los elementos **autor**. Se puede asumir que sólo los libros y los artículos aparecen como elementos de nivel superior en los documentos XML.

```
<!DOCTYPE bibliografía [
 <!ELEMENT libro (título, autor+, año, editor, lugar?)>
 <!ELEMENT artículo (título, autor+, revista, año, número, volumen, páginas?)>
 <!ELEMENT autor (apellidos, nombre) >
 <!ELEMENT título (#PCDATA)>
 ... declaraciones PCDATA similares para el año, el editor, el lugar,
 la revista, el año, el número, el volumen, las páginas, los apellidos y el nombre
] >
```

**Figura 10.15** DTD para los datos bibliográficos.

**10.8** Mostrar la representación en árbol de los datos XML de la Figura 10.1 y la representación del árbol usando las relaciones *nodos* e *hijo* descritas en el Apartado 10.6.2.

**10.9** Considérese la siguiente DTD recursiva:

```
<!DOCTYPE parts [
 <!ELEMENT producto (nombre, infocomponente*)>
 <!ELEMENT infocomponente (producto, cantidad)>
 <!ELEMENT nombre (#PCDATA)>
 <!ELEMENT cantidad (#PCDATA)>
] >
```

- Dese un pequeño ejemplo de datos correspondientes a esta DTD.
- Muéstrese cómo hacer corresponder este DTD con un esquema relacional. Se puede suponer que los nombres de producto son únicos, esto es, cada vez que aparezca un producto, la estructura de sus componentes será la misma.
- Créese un esquema en XML Schema correspondiente a esta DTD.

## Ejercicios

**10.10** Demuéstrese, proporcionando una DTD, cómo representar la relación *libros*, que no está en primera forma normal, del Apartado 9.2 mediante el uso de XML.

**10.11** Escríbanse las siguientes consultas en XQuery, asumiendo la DTD del Ejercicio práctico 10.2.

- Encontrar los nombres de todos los empleados que tienen un hijo cuyo cumpleaños cae en marzo.
- Encontrar aquellos empleados que se examinaron del tipo de materia “mecanografía” en la ciudad “Madrid”.
- Listar todos los tipos de materias de *Emp*.

**10.12** Considérense los datos XML de la Figura 10.2. Supóngase que se desea encontrar los pedidos con dos o más compras del producto con identificador 123. Considérese esta posible solución al problema:

```
for $p in pedidocompra
 where $p/id/producto = 123 and $p/cantidad/producto >= 2
 return $p
```

Explíquese por qué la pregunta puede devolver algunos pedidos con menos de dos compras del producto 123. Dese una versión correcta de esta consulta.

**10.13** Dese una consulta en XQuery para invertir el anidamiento de los datos del Ejercicio 10.10. Esto es, el nivel más externo del anidamiento de la salida debe tener los elementos correspondientes a los autores, y cada uno de estos elementos debe tener anidados los elementos correspondientes a todos los libros escritos por el autor.

- 10.14** Dese la DTD de una representación XML de la información de la Figura 2.29. Créese un tipo de elemento diferente para representar cada relación, pero úsese ID e IDREF para implementar las claves primarias y externas.
- 10.15** Dese una representación en XML Schema de la DTD del Ejercicio 10.14.
- 10.16** Escríbanse las consultas en XQuery del fragmento DTD de bibliografía de la Figura 10.15 para realizar lo siguiente:
- Determinar todos los autores que tienen un libro y un artículo en el mismo año.
  - Mostrar los libros y artículos ordenados por años.
  - Mostrar los libros con más de un autor.
  - Encontrar todos los libros que contengan la palabra “base de datos” en su título y la palabra “Remedios” en el nombre o apellidos del autor.
- 10.17** Dese la versión relacional del esquema de pedidos en XML ilustrado en la Figura 10.2, usando el enfoque descrito en el Apartado 10.6.2.3. Sugírase cómo eliminar la redundancia en el esquema relacional cuando los identificadores determinan funcionalmente a su descripción, y los nombres del comprador y del proveedor determinan funcionalmente a las direcciones del comprador y del proveedor respectivamente.
- 10.18** Escríbanse consultas en SQL/XML para convertir los datos bancarios del esquema relacional que hemos utilizado en capítulos anteriores a los esquemas XML *banco-1* y *banco-2* (para el esquema de *banco-2* se puede asumir que la relación cliente tiene el atributo adicional *id\_cliente*).
- 10.19** Como en el Ejercicio 10.18, escríbanse consultas para convertir los datos bancarios a los esquemas XML *banco-1* y *banco-2*, pero esta vez escribiendo consultas XQuery sobre la base de datos SQL/XML predeterminada a la versión XML.
- 10.20** Una forma de fragmentar un documento XML es emplear XQuery para convertir el esquema a la versión SQL/XML del esquema relacional correspondiente y después usar esta versión en sentido inverso para llenar la relación.  
Como ilustración, dese una consulta XQuery para convertir datos del esquema XML *banco-1* al esquema SQL/XML de la Figura 10.14.
- 10.21** Considérese el esquema XML de ejemplo del Apartado 10.3.2 y escríbanse consultas XQuery para realizar las siguientes tareas:
- Comprobar si se cumple la restricción de clave mostrada en el Apartado 10.3.2.
  - Comprobar si se cumple la restricción keyref mostrada en el Apartado 10.3.2.
- 10.22** Considérese el Ejercicio práctico 10.7 y supóngase que los autores también pueden aparecer como elementos de nivel superior ¿Qué cambio habría que realizar en el esquema relacional?

## Notas bibliográficas

El consorcio W3C (World Wide Web Consortium) actúa como cuerpo normativo para las normas relacionadas con la Web, incluyendo XML básico y todos los lenguajes relacionados con XML tales como XPath, XSLT y XQuery. Se puede obtener en [www.w3c.org](http://www.w3c.org) gran número de informes técnicos que definen las normas relacionadas con XML. Este sitio también contiene tutoriales y punteros a software que implementa las distintas normas. El sitio XML Cover Pages ([www.oasis-open.org/cover/](http://www.oasis-open.org/cover/)) también contiene bastante información sobre XML, incluida la especificación del lenguaje Relax NG para la especificación de esquemas XML.

En el libro de texto Katz et al. [2004] se proporciona un tratamiento detallado de XQuery. Quilt se describe en Chamberlin et al. [2000]. Deutsch et al. [1999] describe el lenguaje XML-QL. La integración de consultas con palabras clave en XML se describe en Florescu et al. [2000] y Amer-Yahia et al. [2004].

Funderburk et al. [2002a], Florescu y Kossmann [1999], Kanne y Moerkotte [2000] y Shanmugasundaram et al. [1999] describen el almacenamiento de datos XML. Eisenberg y Melton [2004a] proporciona una descripción de SQL/XML, mientras que Funderburk et al. [2002b] proporciona descripciones de

SQL/XML y de XQuery. Schning [2001] describe una base de datos diseñada para XML. Véanse los Capítulos 27 a 29 para obtener más información sobre el soporte de XML en bases de datos comerciales. Eisenberg y Melton [2004b] proporcionan una descripción del API XQJ para XQuery.

## Herramientas

Se encuentran disponibles una serie de herramientas de uso público para tratar con XML. El sitio [www.oasis-open.org/cover/](http://www.oasis-open.org/cover/) contiene enlaces a una serie de herramientas software para XML y XSL (incluido XSLT). El sitio Web [www.w3c.org](http://www.w3c.org) de W3C tiene páginas que describen las diferentes normas relacionadas con XML, así como punteros a las herramientas software como implementaciones de lenguajes. Téngase en cuenta que varias de las implementaciones, tales como Galax, son pruebas conceptuales. Aunque pueden servir como herramientas de aprendizaje, no son capaces de manejar bases de datos grandes. Varias bases de datos comerciales, incluidas DB2 de IBM, Oracle y SQL Server de Microsoft, soportan el almacenamiento XML, la publicación empleando varias extensiones de SQL, y las consultas mediante XPath y XQuery.



## Almacenamiento de datos y consultas

Aunque los sistemas de bases de datos proporcionan una visión de alto nivel de los datos, en último término es necesario guardarlos como bits en uno o varios dispositivos de almacenamiento. Una amplia mayoría de sistemas de bases de datos actuales almacenan los datos en discos magnéticos y los extraen a la memoria principal para su procesamiento, o los copian como archivos en cintas u otros dispositivos de copia de seguridad. Las características físicas de los dispositivos de almacenamiento desempeñan un papel importante en el modo en que se almacenan los datos, en especial porque el acceso a un fragmento aleatorio de los datos en el disco resulta mucho más lento que el acceso a la memoria: los accesos al disco invierten decenas de milisegundos, mientras que el acceso a la memoria invierte una décima de microsegundo.

En el Capítulo 11 se comienza con una introducción a los medios físicos de almacenamiento, incluidos los mecanismos para minimizar las posibilidades de pérdida de datos debidas a fallos de los dispositivos. A continuación se describe el modo en que se asignan los registros a los archivos que, a su vez, se asignan a bits del disco. El almacenamiento y la recuperación de los objetos se tratan también en el Capítulo 11.

Muchas consultas sólo hacen referencia a una pequeña parte de los registros de un archivo. Los índices son estructuras que ayudan a localizar rápidamente los registros deseados de una relación, sin tener que examinar todos sus registros. El índice de este libro de texto es un ejemplo de ello aunque está pensado para su empleo por personas, a diferencia de los índices de las bases de datos. En el Capítulo 12 se describen varios tipos de índices utilizados en los sistemas de bases de datos.

Las consultas de los usuarios tienen que ejecutarse sobre el contenido de la base de datos, que reside en los dispositivos de almacenamiento. Suele ser conveniente dividir las consultas en operaciones más pequeñas, que se correspondan aproximadamente con las operaciones del álgebra relacional. En el Capítulo 13 se describe el modo en que se procesan las consultas, presenta los algoritmos para la implementación de las operaciones individuales y esboza el modo en que las operaciones se ejecutan en sincronía para procesar una consulta.

Existen muchas maneras alternativas de procesar cada consulta con costes muy distintos. La optimización de consultas hace referencia al proceso de hallar el método de coste mínimo para evaluar una consulta dada. En el Capítulo 14 se describe este proceso.

# Almacenamiento y estructura de archivos

En los capítulos anteriores se han estudiado los modelos de bases de datos de alto nivel. Por ejemplo, en el nivel *conceptual* o *lógico* se ha presentado una bases de datos del modelo relacional como un conjunto de tablas. En realidad, el modelo lógico de las bases de datos es el mejor nivel para que se centren los *usuarios*. Esto se debe a que el objetivo de los sistemas de bases de datos es simplificar y facilitar el acceso a los datos; no se debe agobiar innecesariamente a quienes utilizan el sistema con los detalles físicos de su implementación.

En este capítulo, no obstante, así como en los Capítulos 12, 13 y 14, se analizan niveles inferiores y se describen diferentes métodos de implementación de los modelos de datos y de los lenguajes presentados en capítulos anteriores. Se comienza con las características de los medios de almacenamiento subyacentes, como sistemas de disco y de cinta. Más adelante se definen varias estructuras de datos que permiten un acceso rápido a los datos. Se consideran varias arquitecturas alternativas, idóneas para diferentes tipos de acceso a los datos. La elección final de la estructura de datos hay que hacerla en función del uso que se espera dar al sistema y de las características de cada máquina concreta.

## 11.1 Visión general de los medios físicos de almacenamiento

La mayoría de sistemas informáticos presentan varios tipos de almacenamientos de datos. Estos medios se clasifican según la velocidad con la que se puede tener acceso a los datos, su coste de adquisición por unidad de datos y su fiabilidad. Entre los medios disponibles habitualmente figuran:

- **Caché.** La caché es el mecanismo de almacenamiento más rápido y costoso. La memoria caché es pequeña; su uso lo gestiona el hardware del sistema informático. Los sistemas de bases de datos no la gestionan.
- **Memoria principal.** El medio de almacenamiento utilizado para los datos con los que se opera es la memoria principal. Las instrucciones máquina operan en la memoria principal. Aunque la memoria principal puede contener muchos megabytes de datos (un PC normal viene con 512 megabytes, como mínimo), o incluso cientos de gigabytes de datos en grandes sistemas servidores suele ser demasiado pequeña (o demasiado cara) para guardar toda la base de datos. El contenido de la memoria principal suele perderse en caso de fallo del suministro eléctrico o de caída del sistema.
- **Memoria flash.** La memoria flash se diferencia de la memoria principal en que los datos no se pierden en caso de que se produzca un corte de la alimentación. La lectura de los datos de la memoria flash emplea menos de cien nanosegundos (un nanosegundo es una milésima de microsegundo), más o menos igual de rápida que la lectura de los datos de la memoria principal. Sin embargo, la escritura de los datos en la memoria flash resulta más complicada (los datos pueden escribirse una vez, lo que tarda de cuatro a diez microsegundos, pero no se pueden sobrescribir).

de manera directa). Para sobrescribir la memoria es necesario borrar simultáneamente todo un banco de memoria, el cual queda preparado para volver a escribir en él. Un inconveniente de la memoria flash es que sólo permite un número limitado de ciclos de borrado, que varía entre diez mil y un millón. Es un tipo de memoria sólo de lectura programable y borrible eléctricamente (*electrically erasable programmable read-only memory, EEPROM*); otras variantes de EEPROM permiten el borrado y reescritura de posiciones individuales de la memoria, pero su empleo no está tan difundido.

La memoria flash se ha hecho popular como sustituta de los discos magnéticos para el almacenamiento de pequeños volúmenes de datos (en el año 2005, normalmente menos de un gigabyte, aunque ya hay memorias flash de mayor capacidad que pueden almacenar varios gigabytes) en los sistemas informáticos de coste reducido que se empotran en otros dispositivos, en computadoras de mano y en otros dispositivos electrónicos digitales como las cámaras digitales (a principios del año 2005, 256 megabytes de memoria flash para cámaras tenían un coste menor de 25 €, mientras que un gigabyte tenía un coste menor de 100 €). La memoria flash se utiliza también en las “llaves USB”, que se pueden conectar a los puertos USB (**Universal Serial Bus**, bus serie universal) de los dispositivos informáticos. Estas llaves USB son muy populares para el transporte datos entre sistemas informáticos (los “disquetes” desempeñaron el mismo papel en su momento, pero han quedado obsoletos debido a su capacidad limitada).

- **Almacenamiento en discos magnéticos.** El principal medio de almacenamiento persistente en conexión es el disco magnético. Generalmente se guarda en ellos toda la base de datos. Para acceder a los datos es necesario trasladarlos desde el disco a la memoria principal. Después de realizar la operación deseada se deben escribir en el disco los datos que se hayan modificado.

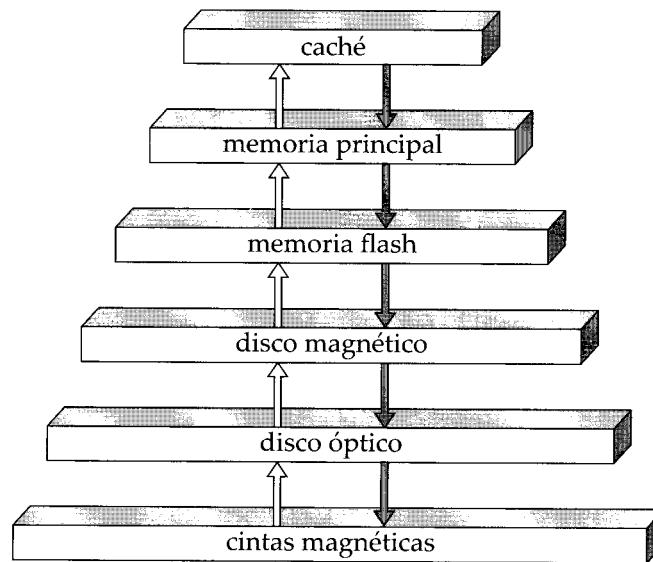
El tamaño de los discos magnéticos actuales varía entre unos pocos gigabytes y 400 gigabytes. Un disco de 250 gigabytes tiene un coste en 2005 alrededor de 160 €. Los dos extremos de este rango han crecido a un ritmo cercano al cincuenta por ciento anual, y cada año se pueden esperar discos de mucha mayor capacidad. El almacenamiento en disco resiste los fallos del suministro eléctrico y las caídas del sistema. Los propios dispositivos de almacenamiento en disco pueden fallar a veces y, en consecuencia, destruir los datos, pero estos fallos se producen con mucha menos frecuencia que las caídas del sistema.

- **Almacenamiento óptico.** La forma más popular de almacenamiento óptico es el disco compacto (*Compact Disk, CD*), que puede almacenar alrededor de 700 megabytes de datos y un tiempo de reproducción de 80 minutos, y el *disco de vídeo digital (Digital Video Disk, DVD)*, que puede almacenar 4,7 u 8,5 gigabytes de datos en cada cara del disco (o hasta 17 gigabytes en un disco de doble cara). También se utiliza el término **disco digital versátil** en lugar de **disco de vídeo digital**, ya que los DVD pueden almacenar cualquier tipo de dato digital, no sólo datos de vídeo. Los datos se almacenan ópticamente en el disco y se leen mediante un láser.

No se puede escribir en los discos ópticos empleados como discos compactos de sólo lectura (CD-ROM) o como discos de vídeo digital de sólo lectura (DVD-ROM), pero se suministran con datos pregrabados. Existen también versiones “para una sola grabación” de los discos compactos (denominados CD-R) y de los discos de vídeo digital (DVD-R y DVD+R) en los que sólo se puede escribir una vez; estos discos también se denominan **de escritura única y lectura múltiple (write-once, read-many, WORM)**. Existen también versiones “para escribir varias veces” de los discos compactos (llamada CD-RW) y de los discos de vídeo digital (DVD-RW, DVD+RW y DVD-RAM), en los que se puede escribir varias veces.

Los **cambiadores automáticos (jukebox)** de discos ópticos contienen varias unidades y numerosos discos que pueden cargarse de manera automática en las diferentes unidades (mediante un brazo robotizado) a petición del usuario.

- **Almacenamiento en cinta.** El almacenamiento en cinta se utiliza principalmente para copias de seguridad y datos archivados. Aunque la cinta magnética es más barata que los discos, el acceso a los datos resulta mucho más lento, ya que hay que acceder secuencialmente desde el comienzo de la cinta. Por este motivo, se dice que el almacenamiento en cinta es de **acceso secuencial**,



**Figura 11.1** Jerarquía de los dispositivos de almacenamiento.

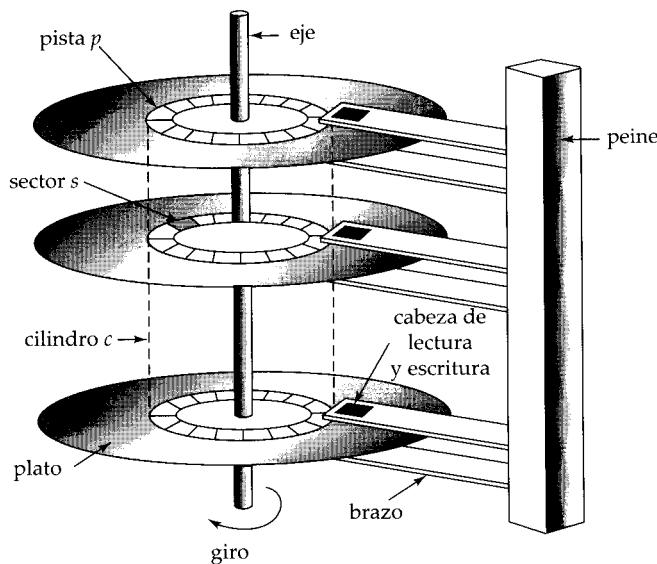
mientras que el almacenamiento en disco es de **acceso directo**, ya que se pueden leer datos de cualquier parte del disco.

Las cintas poseen una capacidad elevada (actualmente se dispone de cintas de 40 a 300 gigabytes) y pueden retirarse de la unidad de lectura, por lo que resultan idóneas para el almacenamiento de archivos con coste reducido. Los cambiadores automáticos de cinta (jukeboxes) se utilizan para guardar conjuntos de datos excepcionalmente grandes, como los datos obtenidos mediante satélite, que pueden ocupar centenares de terabytes ( $10^{12}$  bytes), o incluso petabytes ( $10^{15}$  bytes) en algunos casos.

Los diferentes medios de almacenamiento se pueden organizar de forma jerárquica (Figura 11.1) de acuerdo con su velocidad y su coste. Los niveles superiores resultan caros, pero son rápidos. A medida que se desciende por la jerarquía disminuye el coste por bit, pero aumenta el tiempo de acceso. Este compromiso es razonable; si un sistema de almacenamiento dado fuera a la vez más rápido y menos costoso que otro (en igualdad de resto de condiciones) no habría ninguna razón para utilizar la memoria más lenta y más cara. De hecho, muchos dispositivos de almacenamiento primitivos, como la cinta de papel y las memorias de núcleos de ferrita, se hallan relegados a los museos ahora que la cinta magnética y la memoria de semiconductores son más rápidas y baratas. Las propias cintas magnéticas se utilizaban para guardar los datos activos cuando los discos resultaban costosos y tenían una capacidad de almacenamiento reducida. Hoy en día casi todos los datos activos se almacenan en disco, excepto en los casos excepcionales en que se guardan en cambiadores automáticos de cinta u ópticos.

Los medios de almacenamiento más rápidos (por ejemplo, la caché y la memoria principal) se denominan **almacenamiento primario**. Los medios del siguiente nivel de la jerarquía (por ejemplo, los discos magnéticos) se conocen como **almacenamiento secundario** o **almacenamiento en conexión**. Los medios del nivel inferior de la jerarquía —por ejemplo, la cinta magnética y los cambiadores automáticos de discos ópticos— se denominan **almacenamiento terciario** o **almacenamiento sin conexión**.

Además de la velocidad y del coste de los diferentes sistemas de almacenamiento hay que tener en cuenta la volatilidad del almacenamiento. El **almacenamiento volátil** pierde su contenido cuando se suprime el suministro eléctrico del dispositivo. En la jerarquía mostrada en la Figura 11.1 los sistemas de almacenamiento desde la memoria principal hacia arriba son volátiles, mientras que los sistemas de almacenamiento por debajo de la memoria principal son no volátiles. Los datos se deben escribir en **almacenamiento no volátil** por motivos de seguridad. Este asunto se volverá a tratar en el Capítulo 17.



**Figura 11.2** Mecanismo de disco de cabezas móviles.

## 11.2 Discos magnéticos

Los discos magnéticos proporcionan la mayor parte del almacenamiento secundario de los sistemas informáticos modernos. Aunque la capacidad de los discos ha crecido año tras año, los requisitos de almacenamiento de las grandes aplicaciones también han crecido muy rápido; en algunos casos, más que la capacidad de los discos. Una base de datos de gran tamaño puede necesitar centenares de discos.

### 11.2.1 Características físicas de los discos

Físicamente, los discos son relativamente sencillos (Figura 11.2). Cada **plato** del disco es de forma circular plana. Sus dos superficies están cubiertas por un material magnético en el que se graba la información. Los platos están hechos de metal rígido o de vidrio.

Mientras se utiliza el disco, un motor lo hace girar a una velocidad constante elevada (generalmente sesenta, noventa o ciento veinte revoluciones por segundo, aunque existen discos que giran a doscientas cincuenta revoluciones por segundo). Una cabeza de lectura y escritura está colocada justo encima de la superficie del plato. La superficie del disco se divide a efectos lógicos en **pistas**, que se subdividen en **sectores**. Un **sector** es la unidad mínima de información que se puede leer o escribir en el disco. En los discos actuales, el tamaño de los sectores suele ser de quinientos doce bytes; hay entre cincuenta mil y cien mil pistas en cada plato y de uno a cinco platos por disco. Las pistas internas (las más cercanas al eje) son más cortas, y en los discos actuales, las pistas exteriores contienen más sectores que las internas; suele haber unos quinientos sectores por pista en las pistas internas y alrededor de mil en las externas. Estos números pueden variar de un modelo a otro; los discos de mayor capacidad tienen más sectores por pista y más pistas en cada plato.

La **cabeza de lectura y escritura** guarda la información en los sectores en forma de inversiones de la dirección de magnetización del material magnético.

Cada cara de un plato del disco posee una cabeza de lectura y escritura que se desplaza por el plato para tener acceso a las diferentes pistas. El disco suele contener muchos platos y las cabezas de lectura y escritura de todas las pistas están montadas en un único dispositivo denominado **brazo del disco** y se mueven conjuntamente. El conjunto de los platos del disco montados sobre un mismo eje y de las cabezas montadas en el brazo del disco se denomina **dispositivo cabeza-disco**. Dado que las cabezas de todos los platos se desplazan conjuntamente, cuando la cabeza se halle en la pista  $i$ -ésima de un plato, las restantes también se encontrarán en la pista  $i$ -ésima de sus platos respectivos. Por tanto, el conjunto de las pistas  $i$ -ésimas de todos los platos se denomina **cilindro  $i$ -ésimo**.

Actualmente dominan el mercado los discos con un diámetro de plato de tres pulgadas y media. Tienen un menor coste y tiempos de búsqueda más cortos (debido al menor tamaño) que los discos de mayor tamaño (de hasta catorce pulgadas) que eran habituales en el pasado y, aun así, ofrecen gran capacidad de almacenamiento. Se usan discos de diámetro incluso menor en dispositivos móviles como las computadoras portátiles, las de mano y los reproductores de música de bolsillo.

Las cabezas de lectura y escritura se mantienen tan próximas como resulta posible a la superficie de los discos para aumentar la densidad de grabación. Las cabezas suelen flotar o volar tan sólo a micras de la superficie de cada disco; el giro del disco crea una pequeña corriente de aire y el dispositivo de las cabezas se fabrica de manera que ese flujo de aire mantenga las cabezas flotando rasantes sobre la superficie de los discos. Como las cabezas flotan tan cercanas a la superficie, los platos se deben elaborar con esmero para que sean lisos.

Los choques de las cabezas con la superficie de los platos pueden suponer un problema. Si la cabeza entra en contacto con la superficie del disco, puede arrancar el medio de grabación, lo que destruye los datos que allí hubiera. En los modelos antiguos, el contacto de la cabeza con la superficie hacía que el material arrancado flotase en el aire y se interpusiera entre las cabezas y los platos, lo que causaba más choques; por tanto, la caída de las cabezas podía dar lugar a un fallo de todo el disco. Los dispositivos actuales emplean como medio de grabación una fina capa de metal magnético. Son mucho menos susceptibles de fallar a causa del choque de las cabezas con las superficies de los platos que los antiguos discos recubiertos de óxido.

El **controlador de disco** actúa como interfaz entre el sistema informático y el hardware concreto de la unidad de disco; en los sistemas de disco modernos el controlador se implementa dentro de la unidad de disco. El controlador de disco acepta los comandos de alto nivel de lectura o escritura en un sector dado e inicia las acciones correspondientes, como el desplazamiento del brazo del disco a la pista adecuada y la lectura o escritura real de los datos. Los controladores de disco también añaden **sumas de comprobación** a cada sector en el que se escribe; las sumas de comprobación se calculan a partir de los datos escritos en ese sector. Cuando se vuelve a realizar allí una operación de lectura, se vuelve a calcular esa suma a partir de los datos recuperados y se compara con el valor guardado; si los datos se han deteriorado, resulta muy probable que la suma de comprobación recién calculada no coincida con la guardada. Si se produce un error de este tipo, el controlador volverá a intentar la lectura varias veces; si el error persiste, el controlador indica un fallo de lectura.

Otra labor interesante llevada a cabo por los controladores de disco es la **reasignación de los sectores dañados**. Si el controlador detecta que un sector está dañado cuando se da formato al disco por primera vez, o cuando se realiza un intento de escribir allí, puede reasignar lógicamente el sector a una ubicación física diferente (escogida de entre un grupo de sectores adicionales preparados con esta finalidad). La reasignación se anota en disco o en memoria no volátil y la escritura se realiza en la nueva ubicación.

Los discos se conectan a los sistemas informáticos mediante conexiones de alta velocidad. Las interfaces más comunes para la conexión de los discos a las computadoras personales y a las estaciones de trabajo son: (1) la interfaz ATA (**AT Attachment**) (que es una versión más rápida que la **interfaz electrónica de dispositivos integrados** (IDE, **Integrated Drive Electronics**) usada antiguamente en los PCs de IBM); (2) la nueva versión de ATA: **SATA** (**ATA serie, Serial ATA**; la versión original de ATA se denomina PATA, o Parallel ATA- ATA paralela - para distinguirla de SATA); y (3) la **interfaz de conexión para sistemas informáticos pequeños** (SCSI, **Small Computer System Interconnect**, pronunciado "escasi"). Los grandes sistemas (mainframes) y los sistemas servidores suelen disponer de interfaces más rápidas y caras, como las versiones de alta capacidad de la interfaz SCSI y la interfaz Fibre Channel. Los sistemas de discos externos portátiles utilizan generalmente las interfaces USB o FireWire.

Aunque los discos se suelen conectar directamente a la interfaz de disco de la computadora mediante cables, también pueden hallarse en una ubicación remota y conectarse mediante una red de alta velocidad al controlador de disco. En la arquitectura de **red de área de almacenamiento** (**Storage Area Network, SAN**) se conecta un gran número de discos a varias computadoras servidoras mediante una red de alta velocidad. Los discos suelen disponerse localmente mediante una técnica de organización de almacenamiento denominada **disposición redundante de discos independientes (redundant arrays of independent disks, RAID)** (que se describe en el Apartado 11.3) para proporcionar a los servidores la vista lógica de un solo disco de gran tamaño y muy fiable. La computadora y el subsistema de disco siguen usando las interfaces SCSI y Fibre Channel para comunicarse entre sí, aunque puedan estar sepa-

rados por una red. El acceso remoto a los discos también implica que pueden compartirse entre varias computadoras que pueden ejecutar en paralelo diferentes partes de una misma aplicación. El acceso remoto también supone que los discos que contienen datos importantes se pueden guardar en una sala donde los administradores del sistema pueden supervisarlos y mantenerlos, en lugar de estar dispersos por diferentes locales de la organización.

El **almacenamiento conectado en red** (**Network Attached Storage, NAS**) es una alternativa a SAN. NAS es muy parecido a SAN, salvo que, en lugar de que el almacenamiento en red aparezca como un gran disco, proporciona una interfaz al sistema de archivos que utiliza protocolos de sistemas de archivos en red como NFS o CIFS.

### 11.2.2 Medidas del rendimiento de los discos

Las principales medidas de la calidad de los discos son su capacidad, su tiempo de acceso, su velocidad de transferencia de datos y su fiabilidad.

El **tiempo de acceso** es el tiempo transcurrido desde que se formula una solicitud de lectura o de escritura hasta que comienza la transferencia de datos. Para tener acceso (es decir, para leer o escribir) a los datos de un sector dado del disco, primero se debe desplazar el brazo para que se ubique sobre la pista correcta y luego hay que esperar a que el sector aparezca bajo él debido a la rotación del disco. El tiempo para volver a ubicar el brazo se denomina **tiempo de búsqueda** y aumenta con la distancia que deba recorrer. Los tiempos de búsqueda típicos varían de dos a treinta milisegundos, en función de la distancia de la pista a la posición inicial del brazo. Los discos de menor tamaño tienden a tener tiempos de búsqueda menores, dado que la cabeza tiene que recorrer una distancia menor.

El **tiempo medio de búsqueda** es el promedio de los tiempos de búsqueda medido en una sucesión de solicitudes aleatorias (uniformemente distribuidas). Si todas las pistas tienen el mismo número de sectores y se desprecia el tiempo necesario para que la cabeza comience a moverse y se detenga, se puede demostrar que el tiempo medio de búsqueda es un tercio del peor de los tiempos de búsqueda posibles. Teniendo en cuenta estos factores, el tiempo medio de búsqueda es aproximadamente la mitad del tiempo máximo de búsqueda. Los tiempos medios de búsqueda varían actualmente entre cuatro y diez milisegundos, dependiendo del modelo de disco.

Una vez que la cabeza ha alcanzado la pista deseada, el tiempo de espera hasta que el sector al que hay que acceder aparece bajo la cabeza se denomina **tiempo de latencia rotacional**. Las velocidades rotacionales de los discos varían actualmente entre cinco mil cuatrocientas rotaciones por minuto (noventa rotaciones por segundo) y quince mil rotaciones por minuto (doscientas cincuenta rotaciones por segundo) o, lo que es lo mismo, de cuatro a 11,1 milisegundos por rotación. En promedio, hace falta media rotación del disco para que aparezca bajo la cabeza el comienzo del sector deseado. Por tanto, el **tiempo de latencia medio** del disco es la mitad del tiempo empleado en una rotación completa del disco.

El tiempo de acceso es la suma del tiempo de búsqueda y del tiempo de latencia y varía de ocho a veinte milisegundos. Una vez situado bajo la cabeza el primer sector de los datos, comienza la transferencia. La **velocidad de transferencia de datos** es la velocidad a la que se pueden recuperar datos del disco o guardarlo en él. Los sistemas de disco actuales soportan velocidades máximas de transferencia de veinticinco a cien megabytes por segundo; la velocidad de transferencia medias son significativamente menores que la velocidad de transferencia máxima para las pistas interiores del disco, dado que éstas tienen menos sectores. Por ejemplo, un disco con una velocidad de transferencia máxima de cien megabytes por segundo puede tener una velocidad de transferencia continuada de alrededor de treinta megabytes por segundo en sus pistas internas.

La última medida que se estudiará de entre las utilizadas habitualmente es el **tiempo medio entre fallos**, que indica una medida de la fiabilidad del disco. El tiempo medio entre fallos de un disco (o de cualquier otro sistema) es la cantidad de tiempo que, en promedio, se puede esperar que el sistema funcione de manera continua sin tener ningún fallo. De acuerdo con las afirmaciones de los fabricantes, el tiempo medio entre fallos de los discos actuales varía entre quinientas mil y un millón doscientas mil horas (de cincuenta y siete a ciento treinta y seis años). En la práctica, el tiempo medio entre fallos anunciado se calcula en términos de la probabilidad de fallo cuando el disco es nuevo (este dato significa que, dados mil discos nuevos, si el tiempo medio entre fallos es de un millón doscientas mil horas, en promedio, uno de ellos fallará cada mil doscientas horas). Un tiempo medio entre fallos de un

millón doscientas mil horas no implica que se pueda esperar que el disco funcione ciento treinta y seis años. La mayoría de los discos tienen una esperanza de vida de unos cinco años, y tienen tasas de fallo significativamente más altas en cuanto alcanzan cierta edad.

Las unidades de disco para las máquinas de sobremesa incorporan normalmente las interfaces PATA (Parallel ATA), las cuales proporcionan velocidades de transferencia de ciento treinta y tres megabytes por segundo, y SATA (Serial ATA), capaces de soportar ciento cincuenta megabytes por segundo. Las antiguas interfaces ATA-4 y ATA-5 soportaban velocidades de transferencia de treinta y tres y sesenta y seis megabytes por segundo, respectivamente. Las unidades de disco diseñadas para los sistemas servidores suelen soportar las interfaces Ultra320 SCSI, que permite hasta trescientos veinte megabytes por segundo, y Fibre Channel FC 2Gb, que proporciona tasas de hasta doscientos cincuenta y seis megabytes por segundo. La velocidad de transferencia de cada interfaz se comparte entre todos los discos conectados a la misma (SATA sólo permite que se conecte un disco a cada interfaz).

### 11.2.3 Optimización del acceso a los bloques del disco

Las solicitudes de E/S al disco las generan tanto el sistema de archivos como el gestor de la memoria virtual presente en la mayor parte de los sistemas operativos. Cada solicitud especifica la dirección del disco a la que hay que hacer referencia; esa dirección se expresa como un *número de bloque*. Un **bloque** es una unidad lógica que consiste en un número fijo de sectores contiguos. El tamaño de los bloques varía de quinientos doce bytes a varios kilobytes. Entre el disco y la memoria principal los datos se transfieren por bloques.

El acceso a los datos del disco es varios órdenes de magnitud más lento que el acceso a los datos de la memoria principal. En consecuencia, se han desarrollado diversas técnicas para mejorar la velocidad de acceso a los bloques del disco. El almacenamiento de bloques en la memoria intermedia para satisfacer las solicitudes futuras es una de estas técnicas, la cual se estudia en el Apartado 11.5; aquí se examinan otras.

- **Planificación.** Si hay que transferir varios bloques de un mismo cilindro desde el disco a la memoria principal es posible disminuir el tiempo de acceso solicitando los bloques en el orden en el que pasarán por debajo de las cabezas. Si los bloques deseados se hallan en cilindros diferentes resulta ventajoso solicitar los bloques en un orden que minimice el movimiento del brazo del disco. Los algoritmos de **planificación del brazo del disco** intentan ordenar el acceso a las pistas de manera que se aumente el número de accesos que se puedan procesar. Un algoritmo utilizado con frecuencia es el **algoritmo del ascensor**, que funciona de manera parecida a muchos ascensores. Supóngase que, inicialmente, el brazo se desplaza desde la pista más interna hacia el exterior del disco. Bajo el control del algoritmo del ascensor, el brazo se detiene en cada pista para la que haya una solicitud de acceso, atiende las peticiones para esa pista y continúa desplazándose hacia el exterior hasta que no queden solicitudes pendientes para pistas más externas. En ese punto, el brazo cambia de dirección, se desplaza hacia el interior y vuelve a detenerse en cada pista solicitada hasta que alcanza una pista en la que no haya solicitudes para pistas más cercanas al centro del disco. Entonces cambia de dirección e inicia un nuevo ciclo. Los controladores de disco suelen realizar la labor de reordenar las solicitudes de lectura para mejorar el rendimiento, dado que conocen perfectamente la organización de los bloques del disco, la posición rotacional de los platos y la posición del brazo.
- **Organización de archivos.** Para reducir el tiempo de acceso a los bloques se pueden organizar los bloques del disco de una manera que se corresponda fielmente con la forma en que se espera tener acceso a los datos. Por ejemplo, si se espera tener acceso secuencial a un archivo, se deben guardar secuencialmente en cilindros adyacentes todos los bloques del archivo. Los sistemas operativos más antiguos, como los de IBM para grandes sistemas, ofrecían a los programadores un control detallado de la ubicación de los archivos, lo que permitía reservar un conjunto de cilindros para guardar un archivo. Sin embargo, este control supone una carga para el programador o el administrador del sistema porque que tienen que decidir, por ejemplo, los cilindros que debe asignar a cada archivo. Esto puede exigir una costosa reorganización si se insertan datos en el archivo o se borran de él.

Los sistemas operativos posteriores, como Unix y Windows, ocultan a los usuarios la organización del disco y gestionan la asignación de manera interna. Sin embargo, en el transcurso del tiempo, un archivo secuencial puede quedar **fragmentado**; es decir, sus bloques pueden quedar dispersos por el disco. Para reducir la fragmentación, el sistema puede hacer una copia de seguridad de los datos del disco y restaurarlo completamente. La operación de restauración vuelve a escribir de manera contigua (o casi) los bloques de cada archivo. Algunos sistemas (como las diferentes versiones del sistema operativo Windows) disponen de utilidades que examinan el disco y desplazan los bloques para reducir la fragmentación. Las mejoras de rendimiento obtenidas con estas técnicas son elevadas en algunos casos.

- **Memoria intermedia de escritura no volátil.** Dado que el contenido de la memoria se pierde durante los fallos de suministro eléctrico, hay que guardar en disco la información sobre las actualizaciones de las bases de datos para que superen las posibles caídas del sistema. Debido a esto, el rendimiento de las aplicaciones de bases de datos con gran cantidad de actualizaciones, como los sistemas de procesamiento de transacciones, depende mucho de la velocidad de escritura en el disco.

Se puede utilizar **memoria no volátil de acceso aleatorio** (RAM no volátil, NV-RAM, **Nonvolatile Random-Access Memory**) para acelerar drásticamente la escritura en el disco. El contenido de la NV-RAM no se pierde durante los fallos del suministro eléctrico. Una manera habitual de implementar la NV-RAM es utilizar RAM alimentada por baterías. La idea es que, cuando el sistema de bases de datos (o el sistema operativo) solicita que se escriba un bloque en el disco, el controlador del disco escriba el bloque en una memoria intermedia de NV-RAM y comunique de manera inmediata al sistema operativo que la escritura se completó con éxito. El controlador escribirá los datos en el disco cuando no haya otras solicitudes o cuando se llene la memoria intermedia de NV-RAM. Cuando el sistema de bases de datos solicita la escritura de un bloque, sólo percibe un retraso si la memoria intermedia de NV-RAM se encuentra llena. Durante la recuperación de una caída del sistema se vuelven a escribir en el disco todas las operaciones de escritura pendientes en la memoria intermedia de NV-RAM.

La memoria intermedia NV-RAM se encuentra en algunos discos de altas prestaciones, pero más frecuentemente en los “controladores RAID”; RAID se estudia en el Apartado 11.3.

- **Disco de registro histórico.** Otra manera de reducir las latencias de escritura es utilizar un disco de registro histórico (es decir, un disco destinado a escribir un registro secuencial) de manera muy parecida a la memoria intermedia RAM no volátil. Todos los accesos al disco de registro histórico son secuenciales, lo que básicamente elimina el tiempo de búsqueda y, así, se pueden escribir simultáneamente varios bloques consecutivos, lo que hace que los procesos de escritura en el disco de registro sean varias veces más rápidos que los procesos de escritura aleatorios. Igual que ocurría anteriormente, también hay que escribir los datos en su ubicación verdadera en el disco, pero el disco de registro puede realizar este proceso de escritura más adelante, sin que el sistema de bases de datos tenga que esperar a que se complete. Además, el disco de registro histórico puede reordenar las operaciones de escritura para reducir el movimiento del brazo. Si el sistema se cae antes de que se haya completado alguna operación de escritura en la ubicación real del disco, cuando el sistema se recupera, lee el disco de registro histórico para averiguar las operaciones de escritura que no se han completado y las lleva a cabo.

Los sistemas de archivo que soportan los discos de registro histórico mencionados se denominan **sistemas de archivos de diario**. Los sistemas de archivos de diario se pueden implementar incluso sin un disco de registro histórico independiente, guardando los datos y el registro histórico en el mismo disco. Al hacerlo así se reduce el coste económico, a expensas de un menor rendimiento.

La mayoría de los sistemas de archivos modernos implementan un diario y usan el disco de registro histórico al escribir información interna del sistema de archivos, como la relativa a la asignación de archivos. Los primeros sistemas de archivos permitían reordenar las operaciones de escritura sin emplear disco de registro, y corrían el riesgo de que las estructuras de datos del sistema de archivos del disco se corrompiesen si se producía una caída del sistema. Supóngase, por ejemplo, que un sistema de archivos utiliza una lista enlazada y se inserta un nuevo nodo

al final escribiendo primero los datos del nuevo nodo y actualizando posteriormente el puntero del nodo anterior. Supóngase también que las operaciones de escritura se reordenasen, de forma que se actualizase primero el puntero y que el sistema se cayese antes de escribir el nuevo nodo. El contenido del nodo sería la basura que hubiese anteriormente en el disco, lo que daría lugar a una estructura de datos corrupta.

Para evitar esta posibilidad los primeros sistemas de archivo tenían que realizar una comprobación de consistencia del sistema al reiniciar el sistema para asegurarse de que las estructuras de datos eran consistentes. Si no lo eran, había que tomar medidas adicionales para volver a hacerlas consistentes. Estas comprobaciones daban lugar a grandes retardos en el inicio del sistema después de las caídas, que se agravaron al aumentar la capacidad de las unidades de disco. Los sistemas de archivo de diario permiten un reinicio rápido sin necesidad de esas comprobaciones.

No obstante, las operaciones de escritura realizadas por las aplicaciones no se suelen escribir en el disco del registro histórico. Los sistemas de bases de datos implementan sus propias variantes de registro, que se estudian posteriormente en el Capítulo 17.

## 11.3 RAID

Los requisitos de almacenamiento de datos de algunas aplicaciones (en particular las aplicaciones Web, las de bases de datos y las multimedia) han crecido tan rápidamente que hace falta un gran número de discos para almacenar sus datos, pese a que la capacidad de los discos también ha aumentado mucho.

Tener gran número de discos en el sistema proporciona oportunidades para mejorar la velocidad a la que se pueden leer o escribir los datos si los discos funcionan en paralelo. También se pueden realizar varias operaciones de lectura o escritura independientes simultáneamente. Además, esta configuración ofrece la posibilidad de mejorar la fiabilidad del almacenamiento de datos, ya que se puede guardar información repetida en varios discos. Por tanto, el fallo de uno de los discos no provoca pérdidas de datos.

Para conseguir mayor rendimiento y fiabilidad se han propuesto varias técnicas de organización de los discos, denominadas colectivamente **disposición redundante de discos independientes (RAID, Redundant Array of Independent Disks)**.

En el pasado, los diseñadores de sistemas consideraron los sistemas de almacenamiento compuestos de varios discos pequeños de bajo coste como alternativa económica efectiva al empleo de unidades grandes y caras; el coste por megabyte de los discos más pequeños era menor que el de los de gran tamaño. De hecho, la I de RAID, que ahora significa *independientes*, originalmente representaba *económicos (inexpensive)*. Hoy en día, sin embargo, todos los discos son de pequeño tamaño y los discos de mayor capacidad tienen un menor coste por megabyte. Los sistemas RAID se utilizan por su mayor fiabilidad y por su mejor rendimiento, más que por motivos económicos. Otro motivo fundamental del empleo de RAID es su mayor facilidad de administración y de operación.

### 11.3.1 Mejora de la fiabilidad mediante la redundancia

Considérese en primer lugar la fiabilidad. La probabilidad de que falle, al menos, un disco de un conjunto de  $N$  discos es mucho más elevada que la posibilidad de que falle un único disco concreto. Supóngase que el tiempo medio entre fallos de un disco dado es de cien mil horas o, aproximadamente, once años. Por tanto, el tiempo medio entre fallos de algún disco de una disposición de cien discos será de  $100.000/100 = 1.000$  horas, o cuarenta y dos días, ¡lo que no es mucho! Si sólo se guarda una copia de los datos, cada fallo de disco dará lugar a la pérdida de una cantidad significativa de datos (como ya se estudió en el Apartado 11.2.1). Una frecuencia de pérdida de datos tan elevada resulta inaceptable.

La solución al problema de la fiabilidad es introducir la **redundancia**; es decir, se guarda información adicional que normalmente no se necesita pero que se puede utilizar en caso de fallo de un disco para reconstruir la información perdida. Por tanto, aunque falle un disco, no se pierden datos, por lo que el tiempo medio efectivo entre fallos aumenta, siempre que se cuenten sólo los fallos que dan lugar a pérdida de datos o a su no disponibilidad.

El enfoque más sencillo (pero el más costoso) para la introducción de la redundancia es duplicar todos los discos. Esta técnica se denomina **creación de imágenes** (o, a veces, *creación de sombras*). Cada

unidad lógica consta, por tanto, de dos unidades físicas y cada proceso de escritura se lleva a cabo por duplicado. Si uno de los discos falla se pueden leer los datos del otro. Los datos sólo se perderán si falla el segundo disco antes de que se repare el primero que falló.

El tiempo medio entre fallos (entendiendo fallo como pérdida de datos) de un disco con imagen depende del tiempo medio entre fallos de cada disco y del **tiempo medio de reparación**, que es el tiempo que se tarda (en promedio) en sustituir un disco averiado y en restaurar sus datos. Supóngase que los fallos de los dos discos son *independientes*; es decir, no hay conexión entre el fallo de uno y el del otro. Por tanto, si el tiempo medio entre fallos de un solo disco es de cien mil horas y el tiempo medio de reparación es de diez horas, el **tiempo medio entre pérdidas de datos** de un sistema de discos con imagen es  $100.000^2/(2 * 10) = 500 * 10^6$  horas, o ¡57.000 años! (aquí no se entra en detalle en los cálculos; se proporcionan en las referencias de las notas bibliográficas).

Hay que tener en cuenta que la suposición de independencia de los fallos de los discos no resulta válida. Los fallos en el suministro eléctrico y los desastres naturales como los terremotos, los incendios y las inundaciones pueden dar lugar a daños simultáneos de ambos discos. A medida que los discos envejecen, la probabilidad de fallo aumenta, lo que incrementa la posibilidad de que falle el segundo disco mientras se repara el primero. A pesar de todas estas consideraciones, sin embargo, los sistemas de discos con imagen ofrecen una fiabilidad mucho más elevada que los sistemas de disco único. Hoy en día se dispone de sistemas de discos con imagen con un tiempo medio entre pérdidas de datos de entre quinientas mil y un millón de horas, de cincuenta y cinco a ciento diez años.

Los fallos en el suministro eléctrico son una causa especial de preocupación, dado que tienen lugar mucho más frecuentemente que los desastres naturales. Los fallos en el suministro eléctrico no son un problema si no se está realizando ninguna transferencia de datos al disco cuando tienen lugar. Sin embargo, incluso con la creación de imágenes de los discos, si se hallan en curso procesos de escritura en el mismo bloque en ambos discos y el suministro eléctrico falla antes de que se haya acabado de escribir en ambos bloques, los dos pueden quedar en un estado inconsistente. La solución a este problema es escribir primero una copia y luego la otra, de modo que una de las copias siempre sea consistente. Cuando se vuelve a iniciar el sistema tras un fallo en el suministro eléctrico hacen falta algunas acciones adicionales para recuperar los procesos de escritura incompletos. Este asunto se examina en el Ejercicio práctico 11.2.

### 11.3.2 Mejora del rendimiento mediante el paralelismo

Considérense ahora las ventajas del acceso en paralelo a varios discos. Con la creación de imágenes de los discos la velocidad a la que se pueden procesar las solicitudes de lectura se duplica, dado que pueden enviarse a cualquiera de los discos (siempre y cuando los dos integrantes de la pareja estén operativos, como es el caso habitual). La velocidad de transferencia de cada proceso de lectura es la misma que en los sistemas de disco único, pero el número de procesos de lectura por unidad de tiempo se ha duplicado.

También se puede mejorar la velocidad de transferencia con varios discos **distribuyendo los datos** entre ellos. En su forma más sencilla, la distribución de datos consiste en dividir los bits de cada byte entre varios discos; esto se denomina **distribución en el nivel de bits**. Por ejemplo, si se dispone de una disposición de ocho discos, se puede escribir el bit  $i$  de cada byte en el disco  $i$ . La disposición de ocho discos puede tratarse como un solo disco con sectores que tienen ocho veces el tamaño normal y, lo que es más importante, que tiene ocho veces la velocidad de acceso habitual. En una organización así cada disco toma parte en todos los accesos (de lectura o de escritura) por lo que el número de accesos que pueden procesarse por segundo es aproximadamente el mismo que con un solo disco, pero cada acceso puede octuplicar el número de datos leídos por unidad de tiempo respecto a un solo disco. La distribución en el nivel de bits puede generalizarse a cualquier número de discos que sea múltiplo o divisor de ocho. Por ejemplo, si se utiliza una disposición de cuatro discos, los bits  $i$  y  $4 + i$  de cada byte irán al disco  $i$ .

La **distribución en el nivel de bloques** reparte cada bloque entre varios discos. Trata la disposición de discos como un único disco de gran capacidad y asigna números lógicos a los bloques; se da por supuesto que los números de bloque comienzan en cero. Con una disposición de  $n$  discos, la distribución en el nivel de bloques asigna el bloque lógico  $i$  de la disposición de discos al disco  $(i \bmod n) + 1$ ; utiliza

el bloque físico  $[i/n]$ -ésimo del disco para almacenar el bloque lógico  $i$ . Por ejemplo, con ocho discos, el bloque lógico 0 se almacena el bloque físico 0 del disco 1, mientras que el bloque lógico 11 se almacena en el bloque físico 1 del disco 4. Si se lee un archivo grande, la distribución en el nivel de bloques busca simultáneamente  $n$  bloques en paralelo en los  $n$  discos, lo que permite una gran velocidad de transferencia para lecturas de gran tamaño. Cuando se lee un solo bloque, la velocidad de transferencia de datos es igual que con un único disco, pero los restantes  $n - 1$  discos quedan libres para realizar otras acciones.

La distribución en el nivel de bloques es la forma de distribución de datos más usada. También son posibles otros niveles de distribución, como el de los bytes de cada sector o el de los sectores de cada bloque.

En resumen, en un sistema de discos el paralelismo tiene dos objetivos principales:

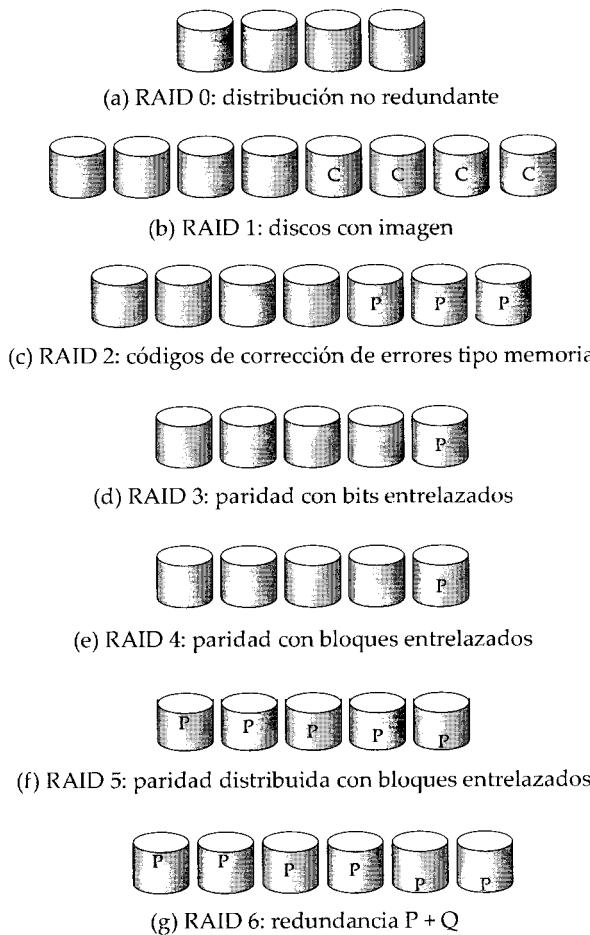
1. Equilibrar la carga de varios accesos de pequeño tamaño (accesos a bloques), de manera que aumente la productividad de ese tipo de accesos.
2. Convertir en paralelos los accesos de gran tamaño para que su tiempo de respuesta se reduzca.

### 11.3.3 Niveles de RAID

La creación de imágenes proporciona gran fiabilidad pero resulta costosa. La distribución proporciona velocidades de transferencia de datos elevadas, pero no mejora la fiabilidad. Varios esquemas alternativos pretenden proporcionar redundancia a un coste menor combinando la distribución de los discos combinada con los bits de “paridad” (que se describen a continuación). Estos esquemas tienen diferentes compromisos entre el coste y el rendimiento y se clasifican en los denominados **niveles de RAID**, como se muestra en la Figura 11.3 (en la figura, P indica los bits para la corrección de errores y C indica una segunda copia de los datos). La figura muestra cuatro discos de datos para todos los niveles, y los discos adicionales se utilizan para guardar la información redundante para la recuperación en caso de fallo.

- **El nivel 0 de RAID** hace referencia a disposiciones de discos con distribución en el nivel de bloques, pero sin redundancia (como la creación de imágenes o los bits de paridad). La Figura 11.3a muestra una disposición de tamaño cuatro.
- **El nivel 1 de RAID** hace referencia a la creación de imágenes del disco con distribución de bloques. La Figura 11.3b muestra una organización con imagen que guarda cuatro discos de datos. Obsérvese que algunos fabricantes emplean el término **nivel 1+0 de RAID** o **nivel 10 de RAID** para referirse a imágenes del disco con distribución, y utilizan el término nivel 1 de RAID para las imágenes del disco sin distribución. Esto último también se puede emplear con disposiciones de discos para dar la apariencia de un disco grande y fiable: si cada disco tiene  $M$  bloques, los bloques lógicos 1 a  $M$  se almacenan en el disco 1;  $M + 1$  a  $2M$  en el segundo, y así sucesivamente, y se crea una imagen de cada disco.<sup>1</sup>
- **El nivel 2 de RAID**, también conocido como organización de códigos de corrección de errores tipo memoria (memory-style-error-correcting-code organization, ECC), emplea bits de paridad. Hace tiempo que los sistemas de memoria utilizan los bits de paridad para la detección y corrección de errores. Cada byte del sistema de memoria puede tener asociado un bit de paridad que registra si el número de bits del byte que valen uno es par (paridad = 0) o impar (paridad = 1). Si alguno de los bits del byte se deteriora (un uno se transforma en cero y viceversa) la paridad del byte se modifica y, por tanto, no coincide con la paridad guardada. Análogamente, si el bit de paridad guardado se deteriora no coincide con la paridad calculada. Por tanto, el sistema de memoria detecta todos los errores que afectan a un número impar de bits de un mismo byte. Los esquemas

1. Obsérvese que algunos fabricantes usan el término RAID 0+1 para referirse a una versión de RAID que utiliza la distribución para crear una disposición RAID 0, y crea una imagen de esa disposición en otra disposición, con la diferencia respecto de RAID 1 de que, si falla un disco, la disposición RAID 0 que contiene el disco queda inutilizable. La disposición con imagen se puede seguir utilizando, así que no hay pérdida de datos. Esta organización es inferior a RAID 1 cuando falla un disco, ya que los demás discos de la disposición RAID 0 se pueden seguir usando en RAID 1, pero quedan inactivos en RAID 0+1.



**Figura 11.3** Niveles de RAID.

de corrección de errores guardan dos o más bits adicionales y pueden reconstruir los datos si se deteriora un solo bit.

La idea de los códigos para la corrección de errores se puede utilizar directamente en las disposiciones de discos mediante la distribución de los bytes entre los diferentes discos. Por ejemplo, el primer bit de cada byte puede guardarse en el disco uno, el segundo en el disco dos, etc., hasta que se guarde el octavo bit en el disco ocho; los bits para la corrección de errores se guardan en discos adicionales.

La Figura 11.3c muestra el esquema de nivel 2. Los discos marcados como *P* guardan los bits para la corrección de errores. Si uno de los discos falla, los restantes bits del byte y los de corrección de errores asociados se pueden leer en los demás discos y se pueden utilizar para reconstruir los datos deteriorados. La Figura 11.3c muestra una disposición de tamaño cuatro; obsérvese que el RAID de nivel 2 sólo necesita tres discos adicionales para los cuatro discos de datos, a diferencia del RAID de nivel 1, que necesitaba cuatro discos adicionales.

- **El nivel 3 de RAID**, organización de paridad con bits entrelazados, mejora respecto al nivel 2 al aprovechar que los controladores de disco, a diferencia de los sistemas de memoria, pueden detectar si los sectores se han leído correctamente, por lo que se puede utilizar un solo bit de paridad para la corrección y para la detección de los errores. La idea es la siguiente: si uno de los sectores se deteriora, se sabe exactamente el sector que es y, para cada uno de sus bits, se puede determinar si es un uno o un cero calculando la paridad de los bits correspondientes de los sectores de los demás discos. Si la paridad de los bits restantes es igual que la paridad guardada, el bit perdido es un cero; en caso contrario, es un uno.

El nivel 3 de RAID es tan bueno como el nivel 2, pero resulta menos costoso en cuanto al número de discos adicionales (sólo tiene un disco adicional), por lo que el nivel 2 no se utiliza en la práctica. La Figura 11.3d muestra el esquema de nivel 3.

El nivel 3 de RAID tiene dos ventajas respecto al nivel 1. Sólo necesita un disco de paridad para varios discos normales, mientras que el nivel 1 necesita un disco imagen por cada disco, por lo que se reduce la necesidad de almacenamiento adicional. Dado que los procesos de lectura y de escritura de cada byte se distribuyen por varios discos, con la distribución de los datos en  $N$  fracciones, la velocidad de transferencia para la lectura o la escritura de un solo bloque multiplica por  $N$  la de una organización RAID de nivel 1 que emplee la distribución entre  $N$  discos. Por otro lado, el nivel 3 de RAID permite un menor número de operaciones de E/S por segundo, ya que todos los discos tienen que participar en cada solicitud de E/S.

- **El nivel 4 de RAID**, organización de paridad con bloques entrelazados, emplea la distribución del nivel de bloques, como RAID 0, y, además, guarda un bloque de paridad en un disco independiente para los bloques correspondientes de los otros  $N$  discos. Este esquema se muestra gráficamente en la Figura 11.3e. Si falla uno de los discos, se puede utilizar el bloque de paridad con los bloques correspondientes de los demás discos para restaurar los bloques del disco averiado.

La lectura de un bloque sólo accede a un disco, lo que permite que los demás discos procesen otras solicitudes. Por tanto, la velocidad de transferencia de datos de cada acceso es menor, pero se pueden ejecutar en paralelo varios accesos de lectura, lo que produce una mayor velocidad global de E/S. Las velocidades de transferencia para los procesos de lectura de gran tamaño son elevadas, dado que se pueden leer todos los discos en paralelo; los procesos de escritura de gran tamaño también tienen velocidades de transferencia elevadas, dado que los datos y la paridad pueden escribirse en paralelo.

Los procesos de escritura independientes de pequeño tamaño, por otro lado, no pueden realizarse en paralelo. La operación de escritura de un bloque tiene que acceder al disco en el que se guarda ese bloque, así como al disco de paridad, ya que hay que actualizar el bloque de paridad. Además, hay que leer tanto el valor anterior del bloque de paridad como el del bloque que se escribe para calcular la nueva paridad. Por tanto, un solo proceso de escritura necesita cuatro accesos a disco: dos para leer los dos bloques antiguos y otros dos para escribir los dos nuevos.

- **El nivel 5 de RAID**, paridad distribuida con bloques entrelazados, mejora respecto al nivel 4 al dividir los datos y la paridad entre todos los  $N + 1$  discos, en vez de guardar los datos en  $N$  discos y la paridad en uno solo. En el nivel 5 todos los discos pueden atender a las solicitudes de lectura, a diferencia del nivel 4 de RAID, en el que el disco de paridad no puede participar, por lo que el nivel 5 aumenta el número total de solicitudes que pueden atenderse en una cantidad de tiempo dada. En cada conjunto de  $N$  bloques lógicos, uno de los discos guarda la paridad y los otros  $N$  guardan los bloques.

La Figura 11.3f muestra esta configuración, en la que las  $P$  están distribuidas entre todos los discos. Por ejemplo, con una disposición de cinco discos, el bloque de paridad, marcado como  $P_k$ , para los bloques lógicos  $4k, 4k + 1, 4k + 2, 4k + 3$  se guarda en el disco  $(k \bmod 5) + 1$ ; los bloques correspondientes de los otros cuatro discos guardan los cuatro bloques de datos  $4k$  to  $4k + 3$ . La siguiente tabla indica la manera en que se disponen los primeros veinte bloques, numerados de 0 to 19, y sus bloques de paridad. El patrón mostrado se repite para los siguientes bloques.

|    |    |    |    |    |
|----|----|----|----|----|
| P0 | 0  | 1  | 2  | 3  |
| 4  | P1 | 5  | 6  | 7  |
| 8  | 9  | P2 | 10 | 11 |
| 12 | 13 | 14 | P3 | 15 |
| 16 | 17 | 18 | 19 | P4 |

Obsérvese que un bloque de paridad no puede guardar la paridad de los bloques del mismo disco, ya que un fallo del disco supondría la pérdida de los datos y de la paridad y, por tanto, no

sería recuperable. El nivel 5 incluye al nivel 4, dado que ofrece mejor rendimiento de lectura y de escritura por el mismo coste, por lo que el nivel 4 no se utiliza en la práctica.

- **El nivel 6 de RAID**, también denominado esquema de redundancia P+Q, es muy parecido al nivel 5, pero guarda información redundante adicional para la protección contra los fallos de disco múltiples. En lugar de utilizar la paridad, el nivel 6 utiliza códigos para la corrección de errores como los de Reed–Solomon (véanse las notas bibliográficas). En el esquema de la Figura 11.3g se guardan dos bits de datos redundantes por cada cuatro bits de datos (a diferencia del único bit de paridad del nivel 5) y el sistema puede tolerar dos fallos del disco.

Finalmente, se debe destacar que se han propuesto varias modificaciones a los esquemas RAID básicos aquí descritos, y que los diferentes fabricantes emplean su propia terminología para las variantes.

#### 11.3.4 Elección del nivel RAID adecuado

Los factores que se deben tener en cuenta para elegir el nivel RAID son:

- El coste económico de los requisitos adicionales de almacenamiento en disco.
- Los requisitos de rendimiento en términos de número de operaciones de E/S.
- El rendimiento cuando falla un disco.
- El rendimiento durante la reconstrucción (esto es, mientras los datos del disco averiado se reconstruyen en un disco nuevo).

El tiempo que se tarda en reconstruir los datos que contenía el disco averiado puede ser significativo, y varía con el nivel RAID utilizado. La reconstrucción resulta más sencilla para el RAID de nivel 1, ya que se pueden copiar los datos de otro disco; para los otros niveles hay que acceder a todos los demás discos de la disposición para reconstruir los datos del disco averiado. El **rendimiento de la reconstrucción** de un sistema RAID puede ser un factor importante si se necesita que los datos estén disponibles ininterrumpidamente, como ocurre en los sistemas de bases de datos de alto rendimiento. Además, dado que el tiempo de reconstrucción puede suponer una parte importante del tiempo de reparación, el rendimiento de la reconstrucción también influye en el tiempo medio entre pérdidas de datos.

El nivel 0 de RAID se utiliza en aplicaciones de alto rendimiento en las que la seguridad de los datos no es crítica. Dado que los niveles 2 y 4 de RAID se subsumen en el 3 y en el 5, respectivamente, la elección de nivel RAID se limita a los niveles restantes. La distribución de bits (nivel 3) es inferior a la distribución de bloques (nivel 5) ya que ésta permite velocidades de transferencia de datos similares para las transferencias de gran tamaño y emplea menos discos para las pequeñas. Para las transferencias de pequeño tamaño, el tiempo de acceso al disco es el factor dominante, por lo que disminuye la ventaja de las operaciones de lectura paralelas. De hecho, el nivel 3 puede presentar un peor rendimiento que el nivel 5 para transferencias de pequeño tamaño, ya que sólo se completan cuando se han encontrado los sectores correspondientes de todos los discos; la latencia media de la disposición de discos se comporta, por tanto, de forma muy parecida a la máxima latencia en el caso de un solo disco, lo que anula las ventajas de la mayor velocidad de transferencia. Muchas implementaciones RAID no soportan actualmente el nivel 6, pero ofrece mayor fiabilidad que el nivel 5 y se puede utilizar en aplicaciones en las que la seguridad de los datos sea muy importante.

La elección entre el nivel 1 y el nivel 5 de RAID es más difícil de tomar. El nivel 1 se utiliza mucho en aplicaciones como el almacenamiento de archivos de registro histórico en sistemas de bases de datos, ya que ofrece el mejor rendimiento para escritura. El nivel 5 tiene menos sobrecarga de almacenamiento que el nivel 1, pero presenta una mayor sobrecarga temporal en las operaciones de escritura. Para las aplicaciones en las que los datos se leen con frecuencia y se escriben raramente, el nivel 5 es la opción preferida.

La capacidad de almacenamiento en disco ha aumentado anualmente más del cincuenta por ciento durante muchos años, y el coste por byte ha disminuido a la misma velocidad. En consecuencia, para muchas aplicaciones existentes de bases de datos con requisitos de almacenamiento moderados, el coste económico del almacenamiento adicional en disco necesario para la creación de imágenes ha pasado a

ser relativamente pequeño (sin embargo, el coste económico adicional, sigue siendo un aspecto significativo para las aplicaciones de almacenamiento intensivo como el almacenamiento de datos de vídeo). La velocidad de acceso ha mejorado mucho más lentamente (del orden de un factor tres en diez años), mientras que el número de operaciones E/S necesarias por segundo se ha incrementado enormemente, especialmente para los servidores de aplicaciones Web.

El nivel 5 de RAID que incrementa el número de operaciones E/S necesarias para escribir un solo bloque lógico, sufre una penalización de tiempo significativa en términos del rendimiento de escritura. El nivel 1 de RAID es, por tanto, la elección adecuada para muchas aplicaciones con requisitos moderados de almacenamiento y grandes de E/S.

Los diseñadores de sistemas RAID también tienen que tomar otras decisiones. Por ejemplo, la cantidad de discos que habrá en la disposición y los bits que debe proteger cada bit de paridad. Cuantos más discos haya en la disposición, mayores serán las velocidades de transferencia de datos, pero el sistema será más caro. Cuantos más bits proteja cada bit de paridad, menor será la necesidad adicional de espacio debida a los bits de paridad, pero habrá más posibilidades de que falle un segundo disco antes de que el primer disco en averiarse esté reparado y de que eso dé lugar a pérdidas de datos.

### 11.3.5 Aspectos hardware

Otro aspecto que debe tenerse en cuenta en la elección de implementaciones RAID es el hardware. RAID se puede implementar sin cambios en el nivel hardware, modificando sólo el software. Estas implementaciones se conocen como **RAID software**. Sin embargo, se pueden obtener ventajas significativas al crear hardware específico para dar soporte a RAID, que se describen a continuación; los sistemas con soporte hardware especial se denominan sistemas **RAID hardware**.

Las implementaciones RAID hardware pueden utilizar RAM no volátil para registrar las operaciones de escritura antes de llevarlas a cabo. En caso de fallo de corriente, cuando el sistema se restaura, recupera la información relativa a las operaciones de escritura incompletas de la memoria RAM no volátil y completa esas operaciones. Sin ese soporte hardware, hay que llevar a cabo trabajo adicional para detectar los bloques que se han escrito parcialmente antes del fallo de corriente (véase el Ejercicio práctico 11.2).

Algunas implementaciones RAID permiten el **intercambio en caliente**; esto es, se pueden retirar los discos averiados y sustituirlos por otros nuevos sin desconectar la corriente. El intercambio en caliente reduce el tiempo medio de reparación, ya que los cambios de disco no necesitan esperar a que se pueda apagar el sistema. De hecho, hoy en día muchos sistemas críticos trabajan con un horario 24 × 7; esto es, funcionan 24 horas al día, 7 días a la semana, lo que no ofrece ningún momento para apagar el sistema y cambiar los discos averiados. Además, muchas implementaciones RAID asignan un disco de repuesto para cada disposición (o para un conjunto de disposiciones de disco). Si falla algún disco, el disco de repuesto se utiliza como sustituto de manera inmediata. En consecuencia, el tiempo medio de reparación se reduce notablemente, lo que minimiza la posibilidad de pérdida de datos. El disco averiado se puede reemplazar cuando resulte más conveniente.

La fuente de alimentación, el controlador de disco o, incluso, la interconexión del sistema en un sistema RAID pueden ser el punto de fallo que detiene el funcionamiento del sistema RAID. Para evitar esta posibilidad, las buenas implementaciones RAID tienen varias fuentes de alimentación redundantes (con baterías de respaldo que les permiten seguir funcionando aunque se corte la corriente). Estos sistemas RAID tienen varias interfaces de disco y varias interconexiones para conectar el sistema RAID con el sistema informático (o con la red de sistemas informáticos). En consecuencia, el fallo de un solo componente no detiene el funcionamiento del sistema RAID.

### 11.3.6 Otras aplicaciones de RAID

Los conceptos de RAID se han generalizado a otros dispositivos de almacenamiento, como los conjuntos de cintas, e incluso a la transmisión de datos por radio. Cuando se aplican a los conjuntos de cintas, las estructuras RAID pueden recuperar datos aunque se haya dañado una de las cintas de la disposición. Cuando se aplican a la transmisión de datos, cada bloque de datos se divide en unidades menores y se transmite junto con una unidad de paridad; si, por algún motivo, no se recibe alguna de las unidades, se puede reconstruir a partir del resto.

## 11.4 Almacenamiento terciario

Puede que en los sistemas de bases de datos de gran tamaño parte de los datos tenga que residir en un almacenamiento terciario. Los dos medios de almacenamiento terciario más habituales son los discos ópticos y las cintas magnéticas.

### 11.4.1 Discos ópticos

Los discos compactos han sido un medio popular de distribución de software, datos multimedia como el sonido y las imágenes, y otra información editada de manera electrónica. Tienen una capacidad de almacenamiento de 640 a 700 megabytes y resultan baratos de producir en masa. Actualmente, los discos de vídeo digital (DVD, Digital Video Disk) han reemplazado a los discos compactos en las aplicaciones que necesitan grandes cantidades de datos. Los discos de formato DVD-5 pueden almacenar 4,7 gigabytes de datos (en una capa de grabación), mientras que los discos de formato DVD-9 pueden almacenar 8,5 gigabytes de datos (en dos capas de grabación). La grabación en las dos caras del disco ofrece capacidades mayores incluso; los formatos DVD-10 y DVD-18, que son las versiones de doble cara de DVD-5 y DVD-9, pueden almacenar 9,4 y 17 gigabytes, respectivamente. Los formatos más recientes, denominados *HD-DVD* y *Blu-Ray DVD* tienen una capacidad significativamente mayor: los discos HD-DVD pueden almacenar de 12 a 30 gigabytes por disco, mientras que los discos Blu-Ray DVD de 25 a 50 gigabytes por disco. Se espera que estén disponibles entre 2005 y 2006.

Las unidades de CD y de DVD presentan tiempos de búsqueda mucho mayores (100 milisegundos es un valor habitual) que las unidades de disco magnético, debido a que el dispositivo de cabezas es más pesado. Las velocidades rotacionales suelen ser menores, aunque las unidades de CD y de DVD más rápidas alcanzan alrededor de tres mil revoluciones por minuto, que son comparables a las velocidades de los discos magnéticos de gama baja. Las velocidades rotacionales de las unidades de CD se correspondían inicialmente con las normas de los CD de sonido; y las velocidades de las unidades de DVD, con las de los DVD de vídeo, pero las unidades actuales rotan a varias veces la velocidad indicada por las normas. Las velocidades de transferencia de datos son algo menores que para los discos magnéticos. Las unidades de CD actuales leen entre 3 y 6 megabytes por segundo; y las de DVD, de 8 a 20 megabytes por segundo. Al igual que las unidades de discos magnéticos, los discos ópticos almacenan más datos en las pistas exteriores y menos en las interiores. La velocidad de transferencia de las unidades ópticas se caracteriza por  $n\times$ , que significa que la unidad soporta transferencias a  $n$  veces la velocidad indicada por la norma; las velocidades de  $50\times$  para CD y de  $16\times$  para DVD son frecuentes hoy en día.

Las versiones de los discos ópticos en las que sólo se puede escribir una vez (CD-R, DVD-R y DVD+R) son populares para la distribución de datos y, en especial, para el almacenamiento de datos con fines de archivo, debido a que tienen una gran capacidad, una vida más larga que los discos magnéticos y pueden retirarse de la unidad y almacenarse en un lugar remoto. Dado que no pueden sobrescribirse, se pueden utilizar para almacenar información que no se deba modificar, como los registros de auditoría. Las versiones en las que se puede escribir más de una vez (CD-RW, DVD-RW, DVD+RW y DVD-RAM) también se emplean para archivo.

Los **cambiadores de discos (jukebox)** son dispositivos que guardan gran número de discos ópticos (hasta varios cientos) y los cargan automáticamente a petición de los usuarios en unas cuantas unidades (usualmente entre una y diez). La capacidad de almacenamiento agregada de estos sistemas puede ser de muchos terabytes. Cuando se accede a un disco, un brazo mecánico lo carga en la unidad desde una estantería (antes hay que volver a colocar en la estantería el disco que se hallara en la unidad). El tiempo de carga y descarga suele ser del orden de segundos (mucho mayor que los tiempos de acceso a disco).

### 11.4.2 Cintas magnéticas

Aunque las cintas magnéticas son relativamente permanentes y pueden albergar grandes volúmenes de datos, resultan lentas en comparación con los discos magnéticos y ópticos. Lo que es aún más importante, la cinta magnética está limitada al acceso secuencial. Por tanto, no puede proporcionar acceso aleatorio para los requisitos de almacenamiento secundario, aunque históricamente, antes del uso de los discos magnéticos, las cintas se utilizaron como medio de almacenamiento secundario.

Las cintas se emplean principalmente para copias de seguridad, para el almacenamiento de la información poco utilizada y como medio sin conexión para la transferencia de información de un sistema a otro. Las cintas también se utilizan para almacenar grandes volúmenes de datos, como los datos de vídeo o de imágenes, a los que no es necesario acceder rápidamente o que son tan voluminosos que su almacenamiento en disco magnético resultaría demasiado caro.

Cada cinta se guarda en una bobina y se enrolla o desenrolla por delante de una cabeza de lectura y escritura. El desplazamiento hasta el punto correcto de la cinta puede tardar segundos o, incluso, minutos, en vez de milisegundos; una vez en posición, sin embargo, las unidades de cinta pueden escribir los datos con densidades y velocidades que se aproximan a las de las unidades de disco. La capacidad varía en función de la longitud y de la anchura de la cinta y de la densidad con la que la cabeza pueda leer y escribir. El mercado ofrece actualmente una amplia variedad de formatos de cinta. La capacidad disponible actualmente varía de unos pocos gigabytes con el formato DAT (**Digital Audio Tape**, cinta de audio digital), de 10 a 40 gigabytes con el formato DLT (**Digital Linear Tape**, cinta lineal digital), de 100 gigabytes en adelante con el formato **Ultrium**, hasta 330 gigabytes con los formatos de cinta de **exploración helicoidal de Ampex**. Las velocidades de transferencia de datos llegan a las decenas de megabytes por segundo.

Las unidades de cinta son bastante fiables, y los buenos sistemas de unidades de cinta realizan una lectura de los datos recién escritos para asegurarse de que se han grabado correctamente. Sin embargo, las cintas presentan límites en cuanto al número de veces que se pueden leer o escribir con fiabilidad.

Los **cambiadores de cintas**, al igual que los cambiadores de discos ópticos, tienen gran número de cintas y unas cuantas unidades en las que se pueden montar esas cintas; se utilizan para guardar grandes volúmenes de datos, que pueden llegar a varios petabytes ( $10^{15}$  bytes) con tiempos de acceso que varían entre los segundos y unos cuantos minutos. Entre las aplicaciones que necesitan estas enormes cantidades de datos están los sistemas de imágenes que reunen los datos de los satélites de teledetección y las grandes videoteca de las emisoras de televisión.

Algunos formatos de cinta (como el formato Accelis) soportan menores tiempos de búsqueda (del orden de decenas de segundos) y están pensadas para las aplicaciones que recuperan información mediante cambiadores de cintas. La mayor parte del resto de los formatos de cinta proporcionan mayores capacidades, a cambio de un acceso más lento; estos formatos resultan idóneos para las copias de seguridad de los datos, en las que las búsquedas rápidas no son importantes.

Las unidades de cinta no han podido competir con la enorme mejora de capacidad de las unidades de disco y la correspondiente reducción del coste de almacenamiento. Aunque el coste de las cintas es bajo, el de las unidades y las bibliotecas de cintas es significativamente superior que el coste de las unidades de disco: una biblioteca de cintas capaz de almacenar algunos terabytes puede costar decenas de millares de euros. Las copias de seguridad en unidades de disco se han convertido en una alternativa rentable a las copias en cinta para gran número aplicaciones.

## 11.5 Acceso al almacenamiento

Cada base de datos se corresponde con varios archivos diferentes que el sistema operativo subyacente mantiene. Esos archivos residen permanentemente en los discos, con copias de seguridad en cinta. Cada archivo está dividido en unidades de almacenamiento de longitud constante denominadas **bloques**, que son las unidades de asignación de almacenamiento y de transferencia de datos. En el Apartado 11.6 se estudian varias maneras de organizar lógicamente los datos en archivos.

Cada bloque puede contener varios elementos de datos. El conjunto exacto de elementos de datos que contiene cada bloque viene determinado por la forma de organización física de los datos que se utilice (véase el Apartado 11.6). En nuestro caso se supone que ningún elemento de datos ocupa dos o más bloques. Esta suposición es realista para la mayor parte de las aplicaciones de procesamiento de datos, como el ejemplo bancario propuesto.

Uno de los principales objetivos del sistema de bases de datos es minimizar el número de transferencias de bloques entre el disco y la memoria. Una manera de reducir el número de accesos al disco es mantener en la memoria principal tantos bloques como sea posible. El objetivo es maximizar la posibilidad de que, cuando se acceda a un bloque, ya se encuentre en la memoria principal y, por tanto, no se necesite acceder al disco.

Dado que no resulta posible mantener en la memoria principal todos los bloques, hay que gestionar la asignación del espacio allí disponible para su almacenamiento. La **memoria intermedia** (buffer) es la parte de la memoria principal disponible para el almacenamiento de las copias de los bloques del disco. Siempre se guarda en el disco una copia de cada bloque, pero esta copia puede ser una versión del bloque más antigua que la de la memoria intermedia. El subsistema responsable de la asignación del espacio de la memoria intermedia se denomina **gestor de la memoria intermedia**.

### 11.5.1 Gestor de la memoria intermedia

Los programas de los sistemas de bases de datos formulan solicitudes (es decir, llamadas) al gestor de la memoria intermedia cuando necesitan bloques del disco. Si el bloque ya se encuentra en la memoria intermedia, el gestor pasa al solicitante la dirección del bloque en la memoria principal. Si el bloque no se halla en la memoria intermedia, asigna en primer lugar espacio al bloque en la memoria intermedia, descartando algún otro, si hace falta, para hacer sitio al nuevo. El bloque descartado sólo se vuelve a escribir en el disco si se ha modificado desde la última vez que se escribió. A continuación, el gestor de la memoria intermedia lee el bloque solicitado en el disco, lo escribe en la memoria intermedia y pasa la dirección del bloque en la memoria principal al solicitante. Las acciones internas del gestor de la memoria intermedia resultan transparentes para los programas que formulan solicitudes de bloques de disco.

Si se está familiarizado con los conceptos de los sistemas operativos, se observará que el gestor de la memoria intermedia no parece ser más que un gestor de memoria virtual, como los que se hallan en la mayor parte de los sistemas operativos. Una diferencia radica en que el tamaño de las bases de datos puede ser mucho mayor que el espacio de direcciones de hardware de la máquina, por lo que las direcciones de memoria no resultan suficientes para direccionar todos los bloques del disco. Además, para dar un buen servicio al sistema de bases de datos, el gestor de la memoria intermedia debe utilizar técnicas más complejas que los esquemas habituales de gestión de la memoria virtual:

- **Estrategia de sustitución.** Cuando no queda espacio libre en la memoria intermedia hay que eliminar un bloque de ésta antes de que se pueda escribir otro nuevo. La mayor parte de los sistemas operativos utilizan un esquema de **menos recientemente utilizado** (Least Recently Used, LRU), en el que se vuelve a escribir en el disco y se elimina de la memoria intermedia el bloque al que se ha hecho referencia menos recientemente. Este sencillo enfoque se puede mejorar para las aplicaciones de bases de datos.
- **Bloques clavados.** Para que el sistema de bases de datos pueda recuperarse de las caídas del sistema (Capítulo 17) hay que restringir las ocasiones en que los bloques se pueden volver a escribir en el disco. Por ejemplo, la mayor parte de los sistemas de recuperación exigen que no se escriban en disco los bloques mientras se esté procediendo a su actualización. Se dice que los bloques que no se permite que se vuelvan a escribir en el disco están **clavados**. Aunque muchos sistemas operativos no permiten trabajar con bloques clavados, esta característica resulta fundamental para los sistemas de bases de datos resistentes a las caídas.
- **Salida forzada de los bloques.** Hay situaciones en las que hace falta volver a escribir los bloques en el disco, aunque no se necesite el espacio de memoria intermedia que ocupan. Este proceso de escritura se denomina **salida forzada** del bloque. Se verá el motivo de las salidas forzadas en el Capítulo 17; en resumen, el contenido de la memoria principal y, por tanto, el de la memoria intermedia se pierden en las caídas, mientras que los datos del disco suelen sobrevivir a ellas.

### 11.5.2 Políticas para la sustitución de la memoria intermedia

El objetivo de las estrategias de sustitución de los bloques de la memoria intermedia es minimizar los accesos al disco. En los programas de propósito general no resulta posible predecir con precisión a qué bloques se hará referencia. Por tanto, el sistema operativo utiliza la pauta anterior de referencias a bloques para predecir las futuras. La suposición que suele hacerse es que es probable que se vuelva a hacer referencia a los bloques a los que se ha hecho referencia recientemente. Por tanto, si hay que sustituir

```

for each tupla b de prestatario do
 for each tupla c de cliente do
 if $b[nombre_cliente] = c[nombre_cliente]$
 then begin
 sea x una tupla definida de la manera siguiente:
 $x[nombre_cliente] := p[nombre_cliente]$
 $x[número_préstamo] := p[número_préstamo]$
 $x[calle_cliente] := c[calle_cliente]$
 $x[ciudad_cliente] := c[ciudad_cliente]$
 incluir la tupla x como parte del resultado de $prestatario \bowtie cliente$
 end
 end
end

```

**Figura 11.4** Procedimiento para calcular la reunión.

un bloque, se sustituye el bloque al que se ha hecho referencia menos recientemente. Este enfoque se denomina **esquema de sustitución de bloques utilizados menos recientemente** (Least Recently Used, LRU).

LRU es un esquema de sustitución aceptable para los sistemas operativos. Sin embargo, los sistemas de bases de datos pueden predecir la pauta de referencias futuras con más precisión que los sistemas operativos. Las peticiones de los usuarios al sistema de bases de datos comprenden varias etapas. El sistema de bases de datos suele poder determinar con antelación los bloques que se necesitarán examinando cada una de las etapas necesarias para llevar a cabo la operación solicitada por el usuario. Por tanto, a diferencia de los sistemas operativos, que deben confiar en el pasado para predecir el futuro, los sistemas de bases de datos pueden tener información relativa, al menos, al futuro a corto plazo.

Para ilustrar la manera en que la información sobre el futuro acceso a los bloques permite mejorar la estrategia LRU considérese el procesamiento de la expresión del álgebra relacional

$$prestatario \bowtie cliente$$

Supóngase que la estrategia escogida para procesar esta solicitud viene dada por el programa en seudocódigo mostrado en la Figura 11.4 (se estudiarán otras estrategias más eficientes en el Capítulo 13).

Supóngase que las dos relaciones de este ejemplo se guardan en archivos diferentes. En este ejemplo se puede ver que, una vez que se haya procesado la tupla de *prestatario*, no vuelve a ser necesaria. Por tanto, una vez completado el procesamiento de un bloque completo de tuplas de *prestatario*, ese bloque ya no se necesita en la memoria principal, aunque se haya utilizado recientemente. Deben darse instrucciones al gestor de la memoria intermedia para que libere el espacio ocupado por el bloque de *prestatario* en cuanto se haya procesado la última tupla. Esta estrategia de gestión de la memoria intermedia se denomina **estrategia de extracción inmediata**.

Considérense ahora los bloques que contienen las tuplas de *cliente*. Hay que examinar cada bloque de las tuplas *cliente* una vez por cada tupla de la relación *prestatario*. Cuando se completa el procesamiento del bloque *cliente*, se sabe que no se accederá nuevamente a él hasta que no se hayan procesado todos los demás bloques de *cliente*. Por tanto, el bloque de *cliente* al que se haya hecho referencia más recientemente será el último bloque al que se vuelva a hacer referencia, y el bloque de *cliente* al que se haya hecho referencia menos recientemente será el bloque al que se vuelva a hacer referencia a continuación. Este conjunto de suposiciones es justo el contrario del que forma la base de la estrategia LRU. En realidad, la estrategia óptima de sustitución de bloques para el procedimiento anterior es la **estrategia más recientemente utilizado** (Most Recently Used, MRU). Si hay que eliminar de la memoria intermedia un bloque de *cliente*, la estrategia MRU escoge el bloque utilizado más recientemente (los bloques en uso no se pueden eliminar).

Para que la estrategia MRU funcione correctamente en el ejemplo propuesto, el sistema debe clavar el bloque de *cliente* que se esté procesando. Después de que se haya procesado la última tupla de *cliente* el bloque se desclava y pasa a ser el bloque utilizado más recientemente.

Además de utilizar la información que pueda tener el sistema respecto de la solicitud que se esté procesando, el gestor de la memoria intermedia puede utilizar información estadística relativa a la probabilidad de que una solicitud haga referencia a una relación concreta. Por ejemplo, el diccionario de datos (como se verá con detalle en el Apartado 11.8) que realiza un seguimiento del esquema lógico de las relaciones y de la información de su almacenamiento físico es una de las partes de la base de datos a la que se tiene acceso con mayor frecuencia. Por tanto, el gestor de la memoria intermedia debe intentar no eliminar de la memoria principal los bloques del diccionario de datos, a menos que se vea obligado a hacerlo. En el Capítulo 12 se estudian los índices de los archivos. Dado que puede que se acceda más frecuentemente al índice del archivo que al propio archivo, el gestor de la memoria intermedia no deberá, en general, eliminar de la memoria principal los bloques del índice si se dispone de alternativas.

La estrategia ideal para la sustitución de bloques necesita información sobre las operaciones de la bases de datos (las que se estén realizando y las que se vayan a realizar en el futuro). No se conoce una sola estrategia que sea adecuada para todas las situaciones posibles. En realidad, un número sorprendentemente grande de bases de datos utiliza LRU, a pesar de sus defectos. Las preguntas prácticas y los ejercicios exploran estrategias alternativas.

La estrategia utilizada por el gestor de la memoria intermedia para la sustitución de los bloques se ve influida por factores distintos del momento en que se volverá a hacer referencia a cada bloque. Si el sistema está procesando de manera concurrente las solicitudes de varios usuarios, puede que el subsistema para el control de concurrencia (Capítulo 16) tenga que posponer ciertas solicitudes para asegurar la conservación de la consistencia de la base de datos. Si se proporciona al gestor de la memoria intermedia información del subsistema de control de concurrencia que indique las solicitudes que se posponen, puede utilizar esa información para modificar su estrategia de sustitución de los bloques. Concretamente, los bloques que necesiten las solicitudes activas (no pospuestas) pueden conservarse en la memoria intermedia a expensas de los que necesiten las solicitudes pospuestas.

El subsistema para la recuperación de caídas (Capítulo 17) impone severas restricciones a la sustitución de bloques. Si se ha modificado un bloque, no se permite que el gestor de la memoria intermedia vuelva a copiar al disco la versión nueva del bloque existente en la memoria intermedia, dado que eso destruiría la versión anterior. Por el contrario, el gestor de bloques debe solicitar permiso del subsistema para la recuperación de caídas antes de escribir cada bloque. Puede que el subsistema para la recuperación de caídas exija que se fuerce la salida de otros bloques antes de conceder autorización al gestor de la memoria intermedia para que escriba el bloque solicitado. En el Capítulo 17 se define con precisión la interacción entre el gestor de la memoria intermedia y el subsistema para la recuperación de caídas.

## 11.6 Organización de los archivos

Los **archivos** se organizan lógicamente como secuencias de registros. Esos registros se corresponden con los bloques del disco. Los archivos constituyen un elemento fundamental de los sistemas operativos, por lo que se supone la existencia de un *sistema de archivos* subyacente. Hay que tomar en consideración diversas maneras de representar los modelos lógicos de datos en términos de los archivos.

Aunque los bloques son de un tamaño fijo determinado por las propiedades físicas del disco y por el sistema operativo, el tamaño de los registros varía. En las bases de datos relacionales, las tuplas de las diferentes relaciones suelen ser de tamaño diferente.

Un enfoque de la correspondencia entre base de datos y archivos es utilizar varios archivos y guardar los registros de la misma longitud en un mismo archivo. Una alternativa es estructurar los archivos de modo que puedan aceptar registros de longitudes diferentes; no obstante, los archivos con registros de longitud fija son más sencillos de implementar que los que tienen registros de longitud variable. Muchas de las técnicas empleadas para los primeros pueden aplicarse a los de longitud variable. Por tanto, se comienza por tomar en consideración los archivos con registros de longitud fija.

### 11.6.1 Registros de longitud fija

A manera de ejemplo, considérese un archivo con registros de *cuentas* de la base de datos bancaria. Cada registro de este archivo se define (en pseudocódigo) de la manera siguiente:

```

type depósito = record
 número_cuenta char(10);
 nombre_sucursal char(22);
 saldo numeric(12,2);
end

```

Si se supone que cada carácter ocupa un byte y que los valores de tipo numeric(12,2) ocupan ocho bytes, el registro de *cuenta* tiene cuarenta bytes de longitud. Un enfoque sencillo es utilizar los primeros cuarenta bytes para el primer registro, los cuarenta bytes siguientes para el segundo, y así sucesivamente (Figura 11.5). Sin embargo, hay dos problemas con este sencillo enfoque:

1. Resulta difícil borrar registros de esta estructura. Hay que llenar el espacio ocupado por el registro que se va a borrar con algún otro registro del archivo, o tener alguna manera de marcar los registros borrados para poder pasarlo por alto.
2. A menos que el tamaño de los bloques sea múltiplo de cuarenta (lo que resulta improbable), algún registro se saltará los límites de los bloques. Es decir, parte del registro se guardará en un bloque y parte en otro. Harán falta, por tanto, dos accesos a bloques para leer o escribir esos registros.

Cuando se borra un registro, se puede desplazar el situado a continuación al espacio ocupado que ocupaba el registro borrado y hacer lo mismo con los demás, hasta que todos los registros situados a continuación del borrado se hayan desplazado hacia delante (Figura 11.6). Este tipo de enfoque necesita desplazar gran número de registros. Puede que fuera más sencillo desplazar simplemente el último registro del archivo al espacio ocupado por el registro borrado (Figura 11.7).

No resulta deseable desplazar los registros para que ocupen el espacio liberado por los registros borrados, ya que para hacerlo se necesitan accesos adicionales a los bloques. Dado que las operaciones de inserción tienden a ser más frecuentes que las de borrado, resulta aceptable dejar libre el espacio

|            |       |                 |     |
|------------|-------|-----------------|-----|
| registro 0 | C-102 | Navacerrada     | 400 |
| registro 1 | C-305 | Collado Mediano | 350 |
| registro 2 | C-215 | Becerril        | 700 |
| registro 3 | C-101 | Centro          | 500 |
| registro 4 | C-222 | Moralzarzal     | 700 |
| registro 5 | C-201 | Navacerrada     | 900 |
| registro 6 | C-217 | Galapagar       | 750 |
| registro 7 | C-110 | Centro          | 600 |
| registro 8 | C-218 | Navacerrada     | 700 |

**Figura 11.5** Archivo que contiene los registros de *cuenta*.

|            |       |                 |     |
|------------|-------|-----------------|-----|
| registro 0 | C-102 | Navacerrada     | 400 |
| registro 1 | C-305 | Collado Mediano | 350 |
| registro 3 | C-101 | Centro          | 500 |
| registro 4 | C-222 | Moralzarzal     | 700 |
| registro 5 | C-201 | Navacerrada     | 900 |
| registro 6 | C-217 | Galapagar       | 750 |
| registro 7 | C-110 | Centro          | 600 |
| registro 8 | C-218 | Navacerrada     | 700 |

**Figura 11.6** El archivo de la Figura 11.5 con el registro 2 borrado y todos los registros desplazados.

|            |       |                 |     |
|------------|-------|-----------------|-----|
| registro 0 | C-102 | Navacerrada     | 400 |
| registro 1 | C-305 | Collado Mediano | 350 |
| registro 8 | C-218 | Navacerrada     | 700 |
| registro 3 | C-101 | Centro          | 500 |
| registro 4 | C-222 | Moralzarzal     | 700 |
| registro 5 | C-201 | Navacerrada     | 900 |
| registro 6 | C-217 | Galapagar       | 750 |
| registro 7 | C-110 | Centro          | 600 |

**Figura 11.7** El archivo de la Figura 11.5 con el registro 2 borrado y el último registro desplazado.

ocupado por los registros borrados y esperar a una inserción posterior antes de volver a utilizar ese espacio. No basta con una simple marca en el registro borrado, ya que resulta difícil hallar el espacio disponible mientras se realiza una inserción. Por tanto, hay que introducir una estructura adicional.

Al comienzo del archivo se asigna cierto número de bytes como **cabecera del archivo**. La cabecera contiene gran variedad de información sobre el archivo. Por ahora, todo lo que hace falta guardar es la dirección del primer registro cuyo contenido se haya borrado. Se utiliza este primer registro para guardar la dirección del segundo registro disponible, y así sucesivamente. De manera intuitiva se pueden considerar estas direcciones guardadas como *punteros*, dado que indican la posición de un registro. Los registros borrados, por tanto, forman una lista enlazada a la que se suele denominar **lista libre**. La Figura 11.8 muestra el archivo de la Figura 11.5, con la lista libre, después de haberse borrado los registros 1, 4 y 6.

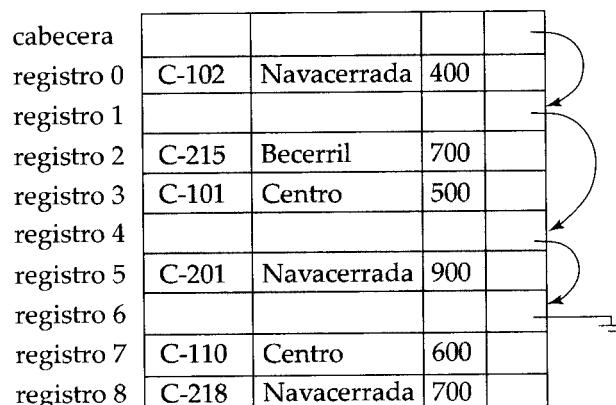
Al insertar un registro nuevo se utiliza el registro al que apunta la cabecera. Se modifica el puntero de la cabecera para que señale al siguiente registro disponible. Si no hay espacio disponible, se añade el nuevo registro al final del archivo.

La inserción y el borrado de archivos con registros de longitud fija son sencillas de implementar, dado que el espacio que deja libre cada registro borrado es exactamente el mismo que se necesita para insertar otro registro. Si se permiten en un archivo registros de longitud variable, esta coincidencia no se mantiene. Puede que el registro insertado no quepa en el espacio liberado por el registro borrado, o que sólo llene una parte.

### 11.6.2 Registros de longitud variable

Los registros de longitud variable surgen de varias maneras en los sistemas de bases de datos:

- Almacenamiento de varios tipos de registros en un mismo archivo.



**Figura 11.8** El archivo de la Figura 11.5 con la lista libre después del borrado de los registros 1, 4 y 6.

- Tipos de registro que permiten longitudes variables para uno o varios de los campos.
  - Tipos de registro que permiten campos repetidos, como los arrays o los multiconjuntos.

Existen diferentes técnicas para implementar los registros de longitud variable.

La **estructura de páginas con ranuras** se utiliza habitualmente para organizar los registros en bloques, y puede verse en la Figura 11.9. Hay una cabecera al principio de cada bloque, que contiene la información siguiente:

1. El número de elementos del registro de la cabecera.
  2. El final del espacio vacío del bloque.
  3. Un *array* cuyas entradas contienen la ubicación y el tamaño de cada registro.

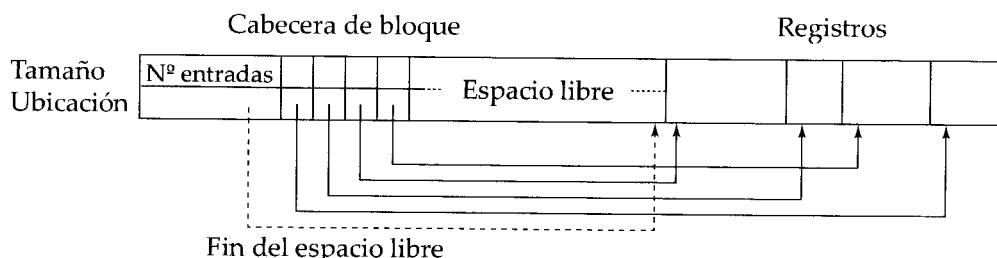
Los registros reales se ubican en el bloque *de manera contigua*, empezando por el final. El espacio libre dentro del bloque es contiguo, entre la última entrada del array de la cabecera y el primer registro. Si se inserta un registro, se le asigna espacio al final del espacio libre y se añade a la cabecera una entrada que contiene su tamaño y su ubicación.

Si se borra un registro, se libera el espacio que ocupa y se da el valor de borrada a su entrada (por ejemplo, se le da a su tamaño el valor de -1). Además, se desplazan los registros del bloque situados antes del registro borrado, de modo que se ocupe el espacio libre creado por el borrado y todo el espacio libre vuelve a hallarse entre la última entrada del array de la cabecera y el primer registro. También se actualiza de manera adecuada el puntero de final del espacio libre de la cabecera. Se puede aumentar o disminuir el tamaño de los registros mediante técnicas parecidas, siempre y cuando quede espacio en el bloque. El coste de trasladar los registros no es demasiado elevado, ya que el tamaño del bloque es limitado: un valor habitual es cuatro kilobytes.

La estructura de páginas con ranuras necesita que no haya punteros que señalen directamente a los registros. Por el contrario, los punteros deben apuntar a la entrada de la cabecera que contiene la ubicación verdadera del registro. Este nivel de tratamiento por la dirección permite que se desplacen los registros para evitar la fragmentación del espacio del bloque al tiempo que permite los punteros indirectos al registro.

Las bases de datos almacenan a menudo datos que pueden ser mucho más grandes que los bloques del disco. Por ejemplo, las imágenes o las grabaciones de sonido pueden tener un tamaño de varios megabytes, mientras que los objetos de vídeo pueden llegar a los gigabytes. Recuérdese que SQL soporta los tipos **blob** y **blob**, que almacenan objetos de gran tamaño de los tipos binario y carácter, respectivamente.

La mayor parte de las bases de datos relacionales limitan el tamaño de los registros para que no superen el tamaño de los bloques, con objeto de simplificar la gestión de la memoria intermedia y del espacio libre. Los objetos de gran tamaño y los campos largos suelen guardarse en archivos especiales (o conjuntos de archivos) en lugar de almacenarse con los otros atributos de pequeño tamaño de los registros en los que aparecen. Los objetos de gran tamaño se suelen representar en organizaciones de archivos de árboles B<sup>+</sup>, que se estudian en el Apartado 12.3.4. Estas organizaciones permiten leer el objeto completo o rangos concretos de bytes del mismo, así como insertar y borrar partes del objeto.



**Figura 11.9** Estructura de páginas con ranuras.

## 11.7 Organización de los registros en archivos

Hasta ahora se ha estudiado la manera en que se representan los registros en la estructura de archivo. Las relaciones son conjuntos de registros. Dado un conjunto de registros, la pregunta siguiente es cómo organizarlos en archivos. A continuación se indican varias maneras de organizar los registros en archivos:

- **Organización de los archivos en montículos.** Se puede colocar cualquier registro en cualquier parte del archivo en que haya espacio suficiente. Los registros no se ordenan. Generalmente sólo hay un archivo para cada relación.
- **Organización secuencial de los archivos.** Los registros se guardan en orden secuencial, según el valor de la “clave de búsqueda” de cada uno. El Apartado 11.7.1 describe esta organización.
- **Organización asociativa (hash) de los archivos.** Se calcula una función de asociación (*hash*) para algún atributo de cada registro. El resultado de la función de asociación especifica el bloque del archivo en que se debe colocar cada registro. El Capítulo 12 describe esta organización; está estrechamente relacionada con las estructuras para la creación de índices que se describen en ese capítulo.

Generalmente se emplea un archivo separado para almacenar los registros de cada relación. No obstante, en cada **organización de archivos en agrupaciones de varias tablas** se pueden guardar en el mismo archivo registros de relaciones diferentes; además, los registros relacionados de las diferentes relaciones se guardan en el mismo bloque, por lo que cada operación de E/S afecta a registros relacionados de todas esas relaciones. Por ejemplo, los registros de dos relaciones se pueden considerar relacionados si casan en una reunión de las dos relaciones. El Apartado 11.7.2 describe esta organización.

### 11.7.1 Organización de archivos secuenciales

Los **archivos secuenciales** están diseñados para el procesamiento eficiente de los registros de acuerdo con un orden basado en alguna clave de búsqueda. La **clave de búsqueda** es cualquier atributo o conjunto de atributos; no tiene por qué ser la clave primaria, ni siquiera una superclave. Para permitir la recuperación rápida de los registros según el orden de la clave de búsqueda, éstos se vinculan mediante punteros. El puntero de cada registro señala al siguiente registro según el orden indicado por la clave de búsqueda. Además, para minimizar el número de accesos a los bloques en el procesamiento de los archivos secuenciales, los registros se guardan físicamente en el orden indicado por la clave de búsqueda, o lo más cercano posible.

La Figura 11.10 muestra un archivo secuencial de registros de *cuenta* tomado del ejemplo bancario propuesto. En ese ejemplo, los registros se guardan de acuerdo con el orden de la clave de búsqueda; en este caso, *nombre\_sucursal*.

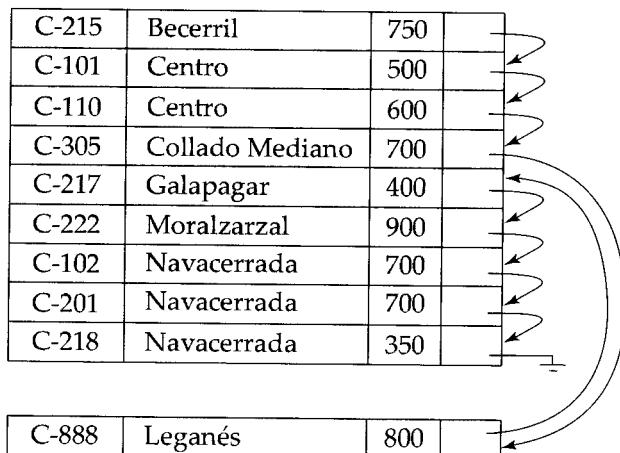
La organización secuencial de archivos permite que los registros se lean de forma ordenada, lo que puede resultar útil para la visualización, así como para ciertos algoritmos de procesamiento de consultas que se estudian en el Capítulo 13.

Sin embargo, resulta difícil mantener el orden físico secuencial a medida que se insertan y se borran registros, dado que resulta costoso desplazar muchos registros como consecuencia de una sola operación de inserción o de borrado. Se puede gestionar el borrado mediante cadenas de punteros, como ya se ha visto. Para la inserción se aplican las reglas siguientes:

1. Localizar el registro del archivo que precede al registro que se va a insertar según el orden de la clave de búsqueda.
2. Si existe algún registro vacío (es decir, un espacio que haya quedado libre después de una operación de borrado) dentro del mismo bloque que ese registro, el registro nuevo se insertará ahí. En caso contrario, el nuevo registro se insertará en un *bloque de desbordamiento*. En cualquier caso, hay que ajustar los punteros para vincular los registros según el orden de la clave de búsqueda.

|       |                 |     |  |
|-------|-----------------|-----|--|
| C-215 | Becerril        | 750 |  |
| C-101 | Centro          | 500 |  |
| C-110 | Centro          | 600 |  |
| C-305 | Collado Mediano | 700 |  |
| C-217 | Galapagar       | 400 |  |
| C-222 | Moralzarzal     | 900 |  |
| C-102 | Navacerrada     | 700 |  |
| C-201 | Navacerrada     | 700 |  |
| C-218 | Navacerrada     | 350 |  |

**Figura 11.10** Archivo secuencial para los registros de *cuenta*.



**Figura 11.11** El archivo secuencial después de una inserción.

La Figura 11.11 muestra el archivo de la Figura 11.10 después de la inserción del registro (C-888, Leganés, 800). La estructura de la Figura 11.11 permite la inserción rápida de registros nuevos, pero obliga a las aplicaciones de procesamiento de archivos secuenciales a procesar los registros en un orden que no coincide con el físico.

Si hay que guardar un número relativamente pequeño de registros en los bloques de desbordamiento, este enfoque funciona bien. Finalmente, no obstante, la correspondencia entre el orden de la clave de búsqueda y el físico puede perderse totalmente, en cuyo caso el procesamiento secuencial acaba siendo mucho menos eficiente. Llegados a este punto, se debe **reorganizar** el archivo de modo que vuelva a estar físicamente en orden secuencial. Estas reorganizaciones resultan costosas y deben realizarse en momentos en los que la carga del sistema sea baja. La frecuencia con la que las reorganizaciones son necesarias depende de la frecuencia de inserción de registros nuevos. En el caso extremo en que rara vez se produzcan inserciones, es posible mantener siempre el archivo en el orden físico correcto. En ese caso, el campo puntero de la Figura 11.10 no es necesario.

## 11.7.2 Organización de archivos en agrupaciones de varias tablas

Muchos sistemas de bases de datos relacionales guardan cada relación en un archivo diferente, de modo que puedan aprovechar completamente el sistema de archivos proporcionado por el sistema operativo. Generalmente las tuplas de cada relación se pueden representar como registros de longitud fija. Por tanto, se puede hacer que las relaciones se correspondan con una estructura de archivos sencilla. Esta implementación sencilla de los sistemas de bases de datos relacionales resulta adecuada para las implementaciones de bajo coste de las bases de datos como, por ejemplo, los sistemas empotrados o los dispositivos portátiles. En estos sistemas el tamaño de la base de datos es pequeño, por lo que se obtiene

| nombre_cliente | número_cuenta |
|----------------|---------------|
| López          | C-102         |
| López          | C-220         |
| López          | C-503         |
| Abril          | C-305         |

**Figura 11.12** La relación *impositor*.

poco provecho de una estructura de archivos avanzada. Además, en esos entornos, es fundamental que el tamaño total del código objeto del sistema de bases de datos sea pequeño. Una estructura de archivos sencilla reduce la cantidad de código necesario para implementar el sistema.

Este enfoque sencillo de la implementación de las bases de datos relacionales resulta menos satisfactorio a medida que aumenta el tamaño de la base de datos. Ya se ha visto que se pueden obtener mejoras en el rendimiento mediante la asignación esmerada de los registros a los bloques y la cuidadosa organización de los propios bloques. Por tanto, resulta evidente que una estructura de archivos más compleja puede resultar beneficiosa, aunque se mantenga la estrategia de guardar cada relación en un archivo diferente.

Sin embargo, muchos sistemas de bases de datos de gran tamaño no utilizan directamente el sistema operativo subyacente para la gestión de los archivos. Por el contrario, se asigna al sistema de bases de datos un archivo de gran tamaño del sistema operativo. En este archivo, el sistema de bases de datos, que también lo administra, guarda todas las relaciones. Para comprender la ventaja de guardar muchas relaciones en un solo archivo considérese la siguiente consulta SQL de la base de datos bancaria:

```
select número_cuenta, nombre_cliente, calle_cliente, ciudad_cliente
from impositor, cliente
where impositor.nombre_cliente = cliente.nombre_cliente
```

Esta consulta calcula una reunión de las relaciones *impositor* y *cliente*. Por tanto, por cada tupla *impositor* el sistema debe encontrar las tuplas de *cliente* con el mismo valor de *nombre\_cliente*. Lo ideal sería poder encontrar estos registros con la ayuda de *índices*, que se estudiarán en el Capítulo 12. Independientemente de la manera en que se encuentren esos registros, hay que transferirlos desde el disco a la memoria principal. En el peor de los casos, cada registro se hallará en un bloque diferente, lo que obligará a efectuar un proceso de lectura de bloque por cada registro necesario para la consulta.

A modo de ejemplo, considérense las relaciones *impositor* y *cliente* de las Figuras 11.12 y 11.13, respectivamente. En la Figura 11.14 se muestra una estructura de archivo diseñada para la ejecución eficiente de las consultas que implican *impositor*  $\bowtie$  *cliente*. Las tuplas *impositor* para cada *nombre\_cliente* se guardan cerca de la tupla *cliente* para el *nombre\_cliente* correspondiente. Esta estructura mezcla las tuplas de dos relaciones, pero permite el procesamiento eficaz de la reunión. Cuando se lee una tupla de la relación *cliente*, se copia del disco a la memoria principal todo el bloque que contiene esa tupla. Dado que las tuplas correspondientes de *impositor* se guardan en el disco cerca de la tupla *cliente*, el bloque que contiene la tupla *cliente* también contiene tuplas de la relación *impositor* necesarias para procesar la consulta. Si un cliente tiene tantas cuentas que los registros de *impositor* no caben en un solo bloque, los registros restantes aparecerán en bloques cercanos.

Una **organización de archivos en agrupaciones de varias tablas** es una organización de archivos, como la mostrada en la Figura 11.14, que almacena registros relacionados de dos o más relaciones en cada bloque. Este tipo de organización de archivos permite leer registros que satisfacen la condición de

| nombre_cliente | calle_cliente | ciudad_cliente |
|----------------|---------------|----------------|
| López          | Mayor         | Arganzuela     |
| Abril          | Preciados     | Valsaín        |

**Figura 11.13** La relación *cliente*.

|       |           |            |  |
|-------|-----------|------------|--|
| López | Mayor     | Arganzuela |  |
| López | C-102     |            |  |
| López | C-220     |            |  |
| López | C-503     |            |  |
| Abril | Preciados | Valsaín    |  |
| Abril | C-305     |            |  |

**Figura 11.14** Estructura de archivo en agrupaciones de varias tablas.

reunión en un solo proceso de lectura de bloques. Por tanto, esta consulta concreta se puede procesar de manera más eficiente.

El empleo de la agrupación de varias tablas en un único archivo ha mejorado el procesamiento de una reunión concreta (*impositor*  $\bowtie$  *cliente*) pero retarda el procesamiento de otros tipos de consulta. Por ejemplo:

```
select *
from cliente
```

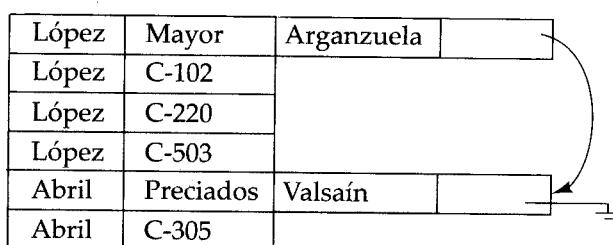
necesita más accesos a los bloques que con el esquema en el que cada relación se guardaba en un archivo diferente. En lugar de aparecer varios registros de *cliente* en un mismo bloque, cada registro se halla en un bloque diferente. En realidad, lograr hallar todos los registros de *cliente* resulta imposible sin alguna estructura adicional. Para encontrar todas las tuplas de la relación *cliente* en la estructura de la Figura 11.14 se pueden enlazar todos los registros de esa relación mediante punteros, tal y como se muestra en la Figura 11.15.

El empleo de la agrupación de varias tablas depende de los tipos de consulta que el diseñador de la base de datos considere más frecuentes. El uso cuidadoso de la agrupación de varias tablas puede producir mejoras de rendimiento significativas en el procesamiento de las consultas.

## 11.8 Almacenamiento con diccionarios de datos

Hasta ahora sólo se ha considerado la representación de las propias relaciones. Un sistema de bases de datos relacionales necesita tener datos *sobre* las relaciones, como puede ser su esquema. Esta información se denomina **diccionario de datos** o **catálogo del sistema**. Entre los tipos de información que debe guardar el sistema figuran los siguientes:

- El nombre de las relaciones.
- El nombre de los atributos de cada relación.
- El dominio y la longitud de los atributos.
- El nombre de las vistas definidas en la base de datos, y la definición de esas vistas.
- Las restricciones de integridad (por ejemplo, las restricciones de las claves).



**Figura 11.15** Estructura de archivo con agrupaciones de varias tablas y cadenas de punteros.

*Metadatos\_relación (nombre\_relación, número\_de\_atributos, organización\_almacenamiento, ubicación)*  
*Metadatos\_atributos (nombre\_atributo, nombre\_relación, tipo\_dominio, posición, longitud)*  
*Metadatos\_usuarios (nombre\_usuario, contraseña\_cifrada, grupo)*  
*Metadatos Índices (nombre Índice, nombre\_relación, tipo Índice, atributos Índice)*  
*Metadatos\_vistas (nombre\_vista, definición)*

**Figura 11.16** Base de datos relacional que representa datos del sistema.

Además, muchos sistemas guardan los siguientes datos de los usuarios del sistema:

- El nombre de los usuarios autorizados.
- La autorización y la información sobre las cuentas de los usuarios.
- Las contraseñas u otra información utilizada para autenticar a los usuarios.

Además, la base de datos puede guardar información estadística y descriptiva sobre las relaciones, como:

- El número de tuplas de cada relación.
- El método de almacenamiento utilizado para cada relación (por ejemplo, con agrupaciones o sin agrupaciones).

El diccionario de datos puede también tener en cuenta la organización del almacenamiento (secuencial, asociativa o en montículos) de las relaciones y la ubicación donde se guarda cada relación:

- Si las relaciones se almacenan en archivos del sistema operativo, el diccionario tendrá en cuenta el nombre del archivo (o archivos) que guarda cada relación.
- Si la base de datos almacena todas las relaciones en un solo archivo, puede que el diccionario tenga en cuenta los bloques que almacenan los registros de cada relación en una estructura de datos como, por ejemplo, una lista enlazada.

En el Capítulo 12, en el que se estudian los índices, se verá que hace falta guardar información sobre cada índice de cada una de las relaciones:

- El nombre del índice.
- El nombre de la relación para la que se crea.
- Los atributos sobre los que se define.
- El tipo de índice formado.

Toda esta información constituye, en efecto, una base de datos en miniatura. Algunos sistemas de bases de datos guardan esta información utilizando código y estructuras de datos especiales. Suele resultar preferible guardar los datos sobre la base de datos en la misma base de datos. Al utilizar la base de datos para guardar los datos del sistema se simplifica la estructura global del sistema y se dedica toda la potencia de la base de datos a obtener un acceso rápido a los datos del sistema.

La elección exacta de la manera de representar los datos del sistema mediante las relaciones debe tomarla el diseñador del sistema. La Figura 11.16 ofrece una representación posible, con las claves primarias subrayadas. En esta representación se da por supuesto que el atributo *atributos Índice* de la relación *Metadatos Índices* contiene una lista de uno o varios atributos, que se pueden representar mediante una cadena de caracteres como “*nombre\_sucursal, ciudad\_sucursal*”. Por tanto, la relación *Metadatos Índices* no está en la primera forma normal; se puede normalizar, pero es probable que la representación anterior sea más eficiente en el acceso. El diccionario de datos se suele almacenar de forma no normalizada para conseguir un acceso rápido.

La organización del almacenamiento y de la ubicación de la propia *Metadatos\_relación* se debe registrar en otro lugar (por ejemplo, en el propio código de la base de datos) ya que esa información es necesaria para encontrar el contenido de *Metadatos\_relación*.

## 11.9 Resumen

- En la mayor parte de los sistemas informáticos hay varios tipos de almacenamiento de datos. Se clasifican según la velocidad con la que se puede acceder a los datos, el coste de adquisición de la memoria por unidad de datos y su fiabilidad. Entre los medios disponibles figuran la memoria caché, la principal, la flash, los discos magnéticos, los discos ópticos y las cintas magnéticas.
- La fiabilidad de los medios de almacenamiento la determinan dos factores: si un corte en el suministro eléctrico o una caída del sistema hace que los datos se pierdan y la probabilidad de fallo físico del dispositivo de almacenamiento.
- Se puede reducir la probabilidad del fallo físico conservando varias copias de los datos. Para los discos se pueden crear imágenes. También se pueden usar métodos más sofisticados como las disposiciones redundantes de discos independientes (RAID). Mediante la distribución de los datos entre los discos estos métodos ofrecen elevados niveles de productividad en los accesos de gran tamaño; la introducción de la redundancia entre los discos se mejora mucho la fiabilidad. Se han propuesto varias organizaciones RAID diferentes, con características de coste, rendimiento y fiabilidad diferentes. Las organizaciones RAID de nivel 1 (creación de imágenes) y de nivel 5 son las más utilizadas.
- Una manera de reducir el número de accesos al disco es conservar todos los bloques posibles en la memoria principal. Dado que allí no se pueden guardar todos, hay que gestionar la asignación del espacio disponible en la memoria principal para el almacenamiento de los bloques. La *memoria intermedia (buffer)* es la parte de la memoria principal disponible para el almacenamiento de las copias de los bloques del disco. El subsistema responsable de la asignación del espacio de la memoria intermedia se denomina *gestor de la memoria intermedia*.
- Los archivos se pueden organizar lógicamente como secuencias de registros asignados a bloques de disco. Un enfoque de la asociación de la base de datos con los archivos es utilizar varios archivos y guardar registros de una única longitud fija en cualquier archivo dado. Una alternativa es estructurar los archivos de modo que puedan aceptar registros de longitud variable. El método de la página con ranuras se usa mucho para manejar los registros de longitud variable en los bloques de disco.
- Dado que los datos se transfieren entre el almacenamiento en disco y la memoria principal en unidades de bloques merece la pena asignar los registros de los archivos a los bloques de modo que cada bloque contenga registros relacionados entre sí. Si se puede tener acceso a varios de los registros deseados utilizando sólo un acceso a bloques, se evitan accesos al disco. Dado que los accesos al disco suelen ser el cuello de botella del rendimiento de los sistemas de bases de datos, la esmerada asignación de registros a los bloques puede ofrecer mejoras significativas del rendimiento.
- El diccionario de datos, también denominado catálogo del sistema, guarda información de los metadatos, que son datos relativos a los datos, como el nombres de las relaciones, el nombre de los atributos y su tipo, la información de almacenamiento, las restricciones de integridad y la información de usuario.

## Términos de repaso

- Medios de almacenamiento físico:
  - Caché.
  - Memoria principal.
  - Memoria flash.
- Disco magnético.
- Almacenamiento óptico.
- Disco magnético

- Plato.
- Discos duros.
- Disquetes.
- Pistas.
- Sectores.
- Cabeza de lectura y escritura.
- Brazo del disco.
- Cilindro.
- Controlador de discos.
- Suma de comprobación.
- Reasignación de sectores defectuosos.
- Medidas de rendimiento de los discos:
  - Tiempo de acceso.
  - Tiempo de búsqueda.
  - Latencia rotacional.
  - Velocidad de transferencia de datos.
  - Tiempo medio entre fallos.
- Bloque de disco.
- Optimización del acceso a bloques de disco.
  - Planificación del brazo.
  - Algoritmo del ascensor.
  - Organización de archivos.
  - Desfragmentación.
  - Memorias intermedias de escritura no volátiles.
  - Memoria no volátil de acceso aleatorio (NV-RAM).
  - Disco del registro histórico.
- Disposición redundante de discos independientes (RAID).
  - Creación de imágenes.
  - Distribución de datos.
  - Distribución en el nivel de bits.
  - Distribución en el nivel de bloques.
- Niveles de RAID:
  - Nivel 0 (distribución de bloques, sin redundancia).
  - Nivel 1 (distribución de bloques, con creación de imágenes).
  - Nivel 3 (distribución de bits, con paridad).
- Nivel 5 (distribución de bloques, con paridad distribuida).
- Nivel 6 (distribución de bloques, con redundancia P+Q).
- Rendimiento de la reconstrucción.
- RAID software.
- RAID hardware.
- Intercambio en caliente.
- Almacenamiento terciario:
  - Discos ópticos.
  - Cintas magnéticas.
  - Cambiadores automáticos.
- Memoria intermedia (buffer).
  - Gestor de la memoria intermedia.
  - Bloques clavados.
  - Salida forzada de bloques.
- Políticas de sustitución de la memoria intermedia:
  - Menos recientemente utilizado (Least Recently Used, LRU).
  - Extracción inmediata.
  - Más recientemente utilizado (Most Recently Used, MRU).
- Archivo.
- Organización de archivos.
  - Cabecera del archivo.
  - Lista libre.
- Registros de longitud variable.
  - Estructura de páginas con ranuras.
- Objetos de gran tamaño.
- Organización de archivos en montículos.
- Organización secuencial de archivos.
- Organización asociativa (hash) de archivos.
- Organización de archivos en agrupaciones de varias tablas.
- Clave de búsqueda.
- Diccionario de datos.
- Catálogo del sistema.

## Ejercicios prácticos

11.1 Considérese la siguiente disposición de los bloques de datos y de paridad de cuatro discos:

| Disco 1 | Disco 2 | Disco 3 | Disco 4  |
|---------|---------|---------|----------|
| $B_1$   | $B_2$   | $B_3$   | $B_4$    |
| $P_1$   | $B_5$   | $B_6$   | $B_7$    |
| $B_8$   | $P_2$   | $B_9$   | $B_{10}$ |
| :       | :       | :       | :        |

$B_i$  representa los bloques de datos;  $P_i$ , los bloques de paridad. El bloque de paridad  $P_i$  es el bloque de paridad para los bloques de datos  $B_{4i-3}$  a  $B_{4i}$ . Indíquense los problemas que puede presentar esta disposición.

- 11.2** Un fallo en el suministro eléctrico que se produzca mientras se escribe un bloque del disco puede dar lugar a que el bloque sólo se escriba parcialmente. Supóngase que se pueden detectar los bloques escritos parcialmente. Un proceso atómico de escritura de bloque es aquél en el que se escribe el bloque entero o no se escribe en absoluto (es decir, no hay procesos de escritura parciales). Propónanse esquemas para conseguir el efecto de los procesos atómicos de escritura de bloques con los siguientes esquemas RAID. Los esquemas deben implicar procesos de recuperación de fallos.

- RAID de nivel 1 (creación de imágenes).
- RAID de nivel 5 (entrelazado de bloques, paridad distribuida).

- 11.3** Dese un ejemplo de expresión de álgebra relacional y de estrategia de procesamiento de consultas en cada una de las situaciones siguientes:

- MRU es preferible a LRU.
- LRU es preferible a MRU.

- 11.4** Considérese el borrado del registro 5 del archivo de la Figura 11.7. Compárense las ventajas relativas de las siguientes técnicas para implementar el borrado:

- Trasladar el registro 6 al espacio que ocupaba el registro 5 y desplazar el registro 7 al espacio ocupado por el registro 6.
- Trasladar el registro 7 al espacio que ocupaba el registro 5.
- Marcar el registro 5 como borrado y no desplazar ningún registro.

- 11.5** Muéstrese la estructura del archivo de la Figura 11.8 después de cada uno de los pasos siguientes:

- Insertar (Galapagar, C-323, 1600).
- Borrar el registro 2.
- Insertar (Galapagar, C-626, 2000).

- 11.6** Considérese una base de datos relacional con dos relaciones:

$$\begin{aligned} \text{asignatura} &(\text{nombre\_asignatura}, \text{aula}, \text{profesor}) \\ \text{matrícula} &(\text{nombre\_asignatura}, \text{nombre\_estudiante}, \text{nota}) \end{aligned}$$

Defínanse ejemplos de estas relaciones para tres asignaturas, en cada una de los cuales se matriculan cinco estudiantes. Dese una estructura de archivos de estas relaciones que utilice la agrupación de varias tablas.

- 11.7** Considérese la siguiente técnica de mapa de bits para realizar el seguimiento del espacio libre de un archivo. Por cada bloque del archivo se mantienen dos bits en el mapa. Si el bloque está lleno entre el 0 y el 30 por ciento, los bits son 00; entre el 30 y el 60 por ciento, 01; entre el 60 y el 90 por ciento, 10; y, por encima del 90 por ciento, 11. Estos mapas de bits se pueden mantener en memoria incluso para archivos de gran tamaño.

- Describábase la manera de mantener actualizado el mapa de bits mientras se insertan y se eliminan registros.
- Describábase la ventaja de la técnica de los mapas de bits frente a las listas libres en la búsqueda de espacio libre y en la actualización de la información.

## Ejercicios

- 11.8** El lector debe indicar los medios de almacenamiento físico disponibles en las computadoras que utiliza habitualmente. Indique la velocidad con la que se puede acceder a los datos en cada medio.
- 11.9** ¿Cómo afecta la reasignación por los controladores de disco de los sectores dañados a la velocidad de recuperación de los datos?

- 11.10** Los sistemas RAID suelen permitir la sustitución de los discos averiados sin interrupción del acceso al sistema. Por tanto, los datos del disco averiado deben reconstruirse y escribirse en el disco de repuesto mientras el sistema se halla en funcionamiento. ¿Con cuál de los niveles RAID es menor la interferencia entre la reconstrucción y el acceso a los discos? Explíquese la respuesta.
- 11.11** Explíquese por qué la asignación de registros a los bloques afecta de manera significativa al rendimiento de los sistemas de bases de datos.
- 11.12** Si es posible, determine la estrategia de gestión de la memoria intermedia del sistema operativo que se ejecuta en su computadora y los mecanismos de que dispone para controlar la sustitución de páginas. Explíquese el modo en que el control sobre la sustitución que proporciona pudiera ser útil para la implementación de sistemas de bases de datos.
- 11.13** En la organización secuencial de los archivos, ¿por qué se utiliza un *bloque* de desbordamiento aunque sólo haya, en ese momento, un registro de desbordamiento?
- 11.14** Indíquense dos ventajas y dos inconvenientes de cada una de las estrategias siguientes para el almacenamiento de bases de datos relacionales:
- Guardar cada relación en un archivo.
  - Guardar varias relaciones (quizá toda la base de datos) en un archivo.
- 11.15** Dese una versión normalizada de la relación *Metadatos\_indices* y explíquese por qué el empleo de la versión normalizada supondría una pérdida de rendimiento.
- 11.16** Si se tienen datos que no se deben perder en un fallo de disco y la escritura de datos es intensiva, ¿cómo se almacenarán esos datos?
- 11.17** En discos de generaciones anteriores el número de sectores por pista era el mismo en todas las pistas. Los discos de las generaciones actuales tienen más sectores por pista en las externas y menos en las internas (ya que son más cortas). ¿Cuál es el efecto de este cambio en cada uno de los tres indicadores principales de la velocidad de los discos?
- 11.18** Los gestores habituales de la memoria intermedia dan por supuesto que cada página es del mismo tamaño y cuesta lo mismo leerla. Considérese un gestor que, en lugar de LRU, utilice la tasa de referencia a objetos, es decir, la frecuencia con que se ha accedido a ese objeto en los últimos  $n$  segundos. Supóngase que se desea almacenar en la memoria intermedia objetos de distinto tamaño y con costes de lectura diferentes (como las páginas Web, cuyo coste de lectura depende del sitio desde el que se busquen). Sugírase cómo puede elegir el gestor la página que debe descartar de la memoria intermedia.

## Notas bibliográficas

Hennessy et al. [2002] es un popular libro de texto de arquitectura de computadoras, que incluye los aspectos de hardware de la memoria intermedia con traducción anticipada, de las cachés y de las unidades de gestión de la memoria. Rosch [2003] presenta una excelente visión general del hardware de las computadoras, incluido un extenso tratamiento de todos los tipos de tecnologías de almacenamiento, como los disquetes, los discos magnéticos, los discos ópticos, las cintas y las interfaces de almacenamiento. Patterson [2004] ofrece un buen estudio sobre el modo en que la mejora de la latencia ha ido por detrás de la mejora del ancho de banda (velocidad de transferencia).

Con el rápido crecimiento de las velocidades de la CPU, las memorias caché ubicadas en la CPU han llegado a ser mucho más rápidas que la memoria principal. Aunque los sistemas de bases de datos no controlan lo que se almacena en la caché, cada vez hay más motivos para organizar los datos en memoria y escribir los programas de forma que se maximice el empleo de la caché. Entre los trabajos en este área figuran Rao y Ross [2000], Ailamaki et al. [2001] y Zhou y Ross [2004].

Las especificaciones de las unidades de disco actuales se pueden obtener de los sitios Web de sus fabricantes, como Hitachi, IBM (que ha vendido recientemente su línea de fabricación de discos a Hitachi), Seagate y Maxtor.

La distribución en los discos se describe en Salem y Garcia-Molina [1986]. Las disposiciones redundantes de discos independientes (RAID) se explican en Patterson et al. [1988] y en Chen y Patterson [1990]. Chen et al. [1994] presenta una excelente revisión de los principios y de la aplicación de RAID. Los códigos de Reed-Solomon se tratan en Pless [1998]. Entre las organizaciones de disco alternativas que proporcionan un alto grado de tolerancia frente a fallos están las descritas en Gray et al. [1990] y en Bitton y Gray [1988]. El sistema de archivos basado en el registro histórico, que transforma en secuencial la escritura en el disco, se describe en Rosenblum y Ousterhout [1991]. Rosenblum y Ousterhout [1991].

En los sistemas que permiten la informática portátil se pueden transmitir los datos de manera reiterada. El medio de transmisión puede considerarse un nivel de la jerarquía de almacenamiento (como los discos transmisores con latencia elevada). Estos problemas se estudian en Acharya et al. [1995]. La gestión de la caché y de las memorias intermedias en la informática portátil se trata en Barbará y Imielinski [1994].

La estructura de almacenamiento de sistemas concretos de bases de datos, como DB2 de IBM, Oracle o SQL Server de Microsoft, se documentan en sus respectivos manuales de sistema.

La gestión de las memorias intermedias se explica en la mayor parte de los textos sobre sistemas operativos, incluido Silberschatz et al. [2001]. Chou y Dewitt [1985] presenta algoritmos para la gestión de la memoria intermedia en los sistemas de bases de datos y describe un método de medida del rendimiento.



# Indexación y asociación

Muchas consultas hacen referencia sólo a una pequeña parte de los registros de un archivo. Por ejemplo, la consulta “Buscar todas las cuentas de la sucursal Pamplona” o “Buscar el saldo del número de cuenta C-101” hace referencia solamente a una fracción de los registros de la relación cuenta. No es eficiente que el sistema tenga que leer cada registro y comprobar que el campo *nombre\_sucursal* contiene el nombre “Pamplona” o el valor C-101 del campo *número\_cuenta*. Lo más adecuado sería que el sistema fuese capaz de localizar directamente esos registros. Para facilitar estas formas de acceso se diseñan estructuras adicionales que se asocian con los archivos.

## 12.1 Conceptos básicos

Un índice para un archivo del sistema funciona como el índice de este libro. Si se va a buscar un tema (especificado por una palabra o una frase) se puede buscar en el índice al final del libro, encontrar las páginas en las que aparece y después leerlas para encontrar la información buscada. Las palabras del índice están ordenadas alfabéticamente, lo cual facilita la búsqueda. Además, el índice es mucho más pequeño que el libro, con lo que se reduce aún más el esfuerzo necesario para encontrar las palabras en cuestión.

Los índices de los sistemas de bases de datos juegan el mismo papel que los índices de los libros en las bibliotecas. Por ejemplo, para recuperar un registro *cuenta* dado su número de cuenta, el sistema de bases de datos buscaría en un índice para encontrar el bloque de disco en que se localice el registro correspondiente, y entonces extraería ese bloque de disco para obtener el registro *cuenta*.

Almacenar una lista ordenada de números de cuenta no funcionaría bien en bases de datos muy grandes con millones de cuentas, ya que el propio índice sería muy grande; más aún, incluso el mantener ordenado el índice reduce el tiempo de búsqueda, por lo que encontrar una cuenta podría consumir mucho tiempo. En su lugar se usan técnicas más sofisticadas de indexación. Algunas de estas técnicas se estudiarán más adelante.

Hay dos tipos básicos de índices:

- **Índices ordenados.** Estos índices están basados en una disposición ordenada de los valores.
- **Índices asociativos.** Estos índices están basados en una distribución uniforme de los valores a través de una serie de cajones (*buckets*). El valor asignado a cada cajón está determinado por una función, llamada *función de asociación (hash function)*.

Se considerarán varias técnicas de indexación y asociación. Ninguna de ellas es la mejor. Sin embargo, para cada aplicación específica de bases de datos existe una técnica más apropiada. Cada una de ellas debe ser valorada según los siguientes criterios:

- **Tipos de acceso.** Los tipos de acceso que se soportan eficazmente. Estos tipos podrían incluir la búsqueda de registros con un valor concreto en un atributo, o la búsqueda de los registros cuyos atributos contengan valores en un rango especificado.
- **Tiempo de acceso.** El tiempo que se tarda en hallar un determinado elemento de datos, o conjunto de elementos, usando la técnica en cuestión.
- **Tiempo de inserción.** El tiempo empleado en insertar un nuevo elemento de datos. Este valor incluye el tiempo utilizado en hallar el lugar apropiado donde insertar el nuevo elemento de datos, así como el tiempo empleado en actualizar la estructura del índice.
- **Tiempo de borrado.** El tiempo empleado en borrar un elemento de datos. Este valor incluye el tiempo utilizado en hallar el elemento a borrar, así como el tiempo empleado en actualizar la estructura del índice.
- **Espacio adicional requerido.** El espacio adicional ocupado por la estructura del índice. Como normalmente la cantidad necesaria de espacio adicional suele ser moderada, es razonable sacrificar el espacio para alcanzar un rendimiento mejor.

A menudo se desea tener más de un índice por archivo. Por ejemplo se puede buscar un libro según el autor, la materia o el título.

Los atributos o conjunto de atributos usados para buscar en un archivo se llaman **claves de búsqueda**. Hay que observar que esta definición de *clave* difiere de la usada en *clave primaria*, *clave candidata* y *superclave*. Este doble significado de *clave* está (desafortunadamente) muy extendido en la práctica. Usando el concepto de clave de búsqueda se determina que, si existen varios índices para un archivo, existirán varias claves de búsqueda.

## 12.2 Índices ordenados

Para permitir un acceso directo rápido a los registros de un archivo se puede usar una estructura de índice. Cada estructura de índice está asociada con una clave de búsqueda concreta. Al igual que en el catálogo de una biblioteca, un índice almacena de manera ordenada los valores de las claves de búsqueda, y asocia a cada clave los registros que contienen esa clave de búsqueda.

Los registros en el archivo indexado pueden estar a su vez almacenados siguiendo un orden, semejante a como los libros están ordenados en una biblioteca por algún atributo como el número decimal Dewey. Un archivo puede tener varios índices según diferentes claves de búsqueda. Si el archivo que contiene los registros está ordenado secuencialmente, el índice cuya clave de búsqueda especifica el orden secuencial del archivo es el **índice con agrupación** (clustering index). Los índices con agrupación también se llaman **índices primarios**; el término índice primario se usa algunas veces para hacer alusión a un índice según una clave primaria pero en realidad se puede usar sobre cualquier clave de búsqueda. La clave de búsqueda de un índice con agrupación es normalmente la clave primaria, aunque no es así necesariamente. Los índices cuyas claves de búsqueda especifican un orden diferente del orden secuencial del archivo se llaman **índices sin agrupación** (non clustering indices) o **secundarios**. Se suelen usar los términos “agrupado” y “no agrupado” en lugar de “con agrupación” y “sin agrupación”.

Desde el Apartado 12.2.1 hasta el 12.2.3 se asume que todos los archivos están ordenados secuencialmente según alguna clave de búsqueda. Estos archivos con un índice con agrupación según la clave de búsqueda se llaman **archivos secuenciales indexados**. Representan uno de los esquemas de índices más antiguos usados por los sistemas de bases de datos. Se usan en aquellas aplicaciones que demandan un procesamiento secuencial del archivo completo, así como un acceso directo a sus registros. En el Apartado 12.2.4 se tratan los índices secundarios.

En la Figura 12.1 se muestra un archivo secuencial de los registros *cuenta* tomados del ejemplo bancario. En esta figura, los registros están almacenados según el orden de la clave de búsqueda: *nombre\_sucursal*.

|       |           |     |  |
|-------|-----------|-----|--|
| C-217 | Barcelona | 750 |  |
| C-101 | Daimiel   | 500 |  |
| C-110 | Daimiel   | 600 |  |
| C-215 | Madrid    | 700 |  |
| C-102 | Pamplona  | 400 |  |
| C-201 | Pamplona  | 900 |  |
| C-218 | Pamplona  | 700 |  |
| C-222 | Reus      | 700 |  |
| C-305 | Ronda     | 350 |  |

**Figura 12.1** Archivo secuencial para los archivos de cuenta.

### 12.2.1 Índices densos y dispersos

Un **registro índice** o **entrada del índice** consiste en un valor de la clave de búsqueda y punteros a uno o más registros con ese valor de la clave de búsqueda. El puntero a un registro consiste en el identificador de un bloque de disco y un desplazamiento en el bloque de disco para identificar el registro dentro del bloque.

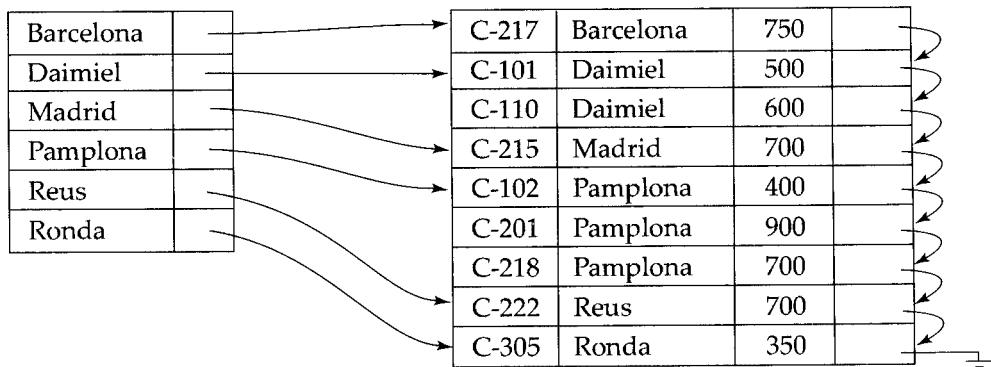
Existen dos clases de índices ordenados que se pueden usar:

- **Índice denso.** Aparece un registro índice por cada valor de la clave de búsqueda en el archivo. En un índice denso con agrupación el registro índice contiene el valor de la clave y un puntero al primer registro con ese valor de la clave de búsqueda. El resto de registros con el mismo valor de la clave de búsqueda se almacenan consecutivamente después del primer registro, dado que, ya que el índice es con agrupación, los registros se ordenan sobre la misma clave de búsqueda.  
Las implementaciones de índices densos pueden almacenar una lista de punteros a todos los registros con el mismo valor de la clave de búsqueda; esto no es esencial para los índices con agrupación.
- **Índice disperso.** Sólo se crea un registro índice para algunos de los valores. Al igual que en los índices densos, cada registro índice contiene un valor de la clave de búsqueda y un puntero al primer registro con ese valor de la clave. Para localizar un registro se busca la entrada del índice con el valor más grande que sea menor o igual que el valor que se está buscando. Se empieza por el registro apuntado por esa entrada del índice y se continúa con los punteros del archivo hasta encontrar el registro deseado.

Las Figuras 12.2 y 12.3 son ejemplos de índices densos y dispersos, respectivamente, para el archivo *cuenta*. Supóngase que se desea buscar los registros de la sucursal Pamplona. Mediante el índice denso de la Figura 12.2, se sigue el puntero que va directo al primer registro de Pamplona. Se procesa el registro y se sigue el puntero en ese registro hasta localizar el siguiente registro según el orden de la clave de búsqueda (*nombre\_sucursal*). Se continuaría procesando registros hasta encontrar uno cuyo nombre de sucursal fuese distinto de Pamplona. Si se usa un índice disperso (Figura 12.3), no se encontraría entrada del índice para “Pamplona”. Como la última entrada (en orden alfabético) antes de “Pamplona” es “Madrid”, se sigue ese puntero. Entonces se lee el archivo *cuenta* en orden secuencial hasta encontrar el primer registro Pamplona, y se continúa procesando desde este punto.

Como se ha visto, generalmente es más rápido localizar un registro si se usa un índice denso en vez de un índice disperso. Sin embargo, los índices dispersos tienen algunas ventajas sobre los índices densos, como el utilizar un espacio más reducido y un mantenimiento adicional menor para las inserciones y borrados.

Existe un compromiso que el diseñador del sistema debe mantener entre el tiempo de acceso y el espacio adicional requerido. Aunque la decisión sobre este compromiso depende la aplicación en particular, un buen compromiso es tener un índice disperso con una entrada del índice por cada bloque. La razón por la cual este diseño alcanza un buen compromiso reside en que el mayor coste de procesar

**Figura 12.2** Índice denso.

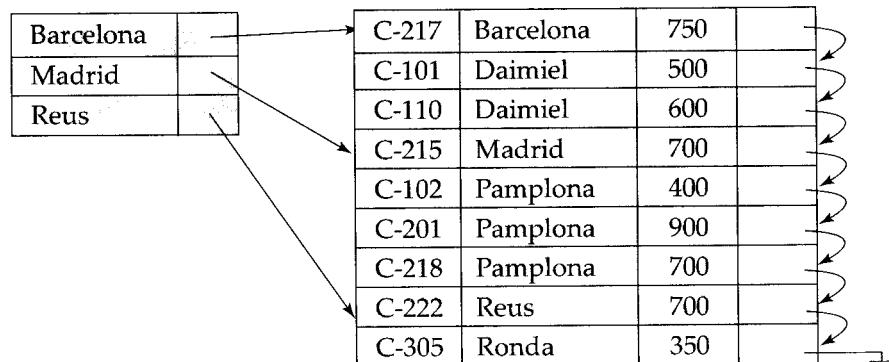
una petición en la base de datos es el empleado en traer un bloque de disco a la memoria. Una vez el bloque en memoria, el tiempo en examinarlo es prácticamente inapreciable. Usando este índice disperso se localiza el bloque que contiene el registro solicitado. De este manera, a menos que el registro esté en un bloque de desbordamiento (véase el Apartado 12.7.1) se minimizan los accesos a bloques mientras el tamaño del índice se mantiene (y así, el espacio adicional requerido) tan pequeño como sea posible.

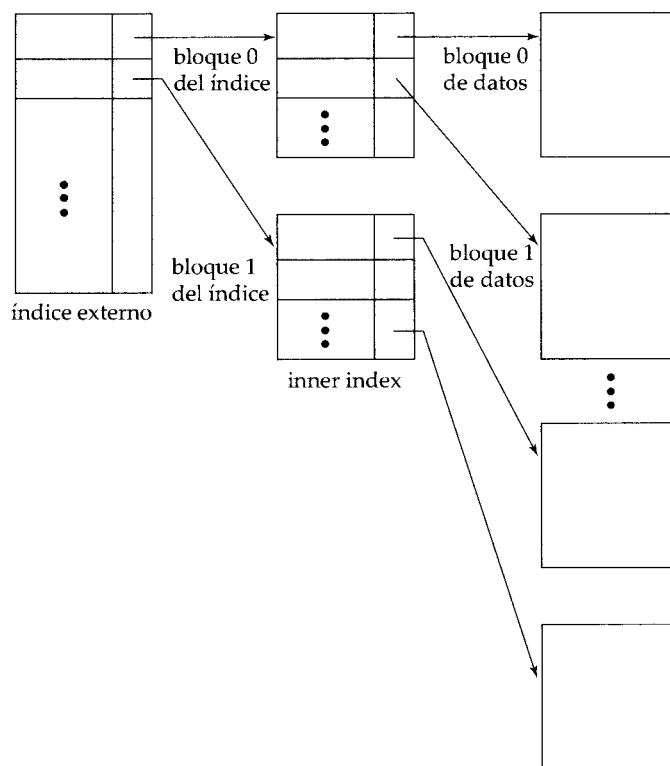
Para generalizar la técnica anterior hay que tener en cuenta si los registros de una clave de búsqueda ocupan varios bloques. Es fácil modificar el esquema para acomodarse a esta situación.

### 12.2.2 Índices multinivel

Aunque se use un índice disperso, puede que el propio índice sea demasiado grande para un procesamiento eficiente. En la práctica no es excesivo tener un archivo con 100.000 registros, con 10 registros almacenados en cada bloque. Si se dispone de un registro índice por cada bloque, el índice tendrá 10.000 registros. Como los registros índices son más pequeños que los registros de datos, se puede suponer que caben 100 registros índices en un bloque. Por tanto el índice ocuparía 100 bloques. Estos índices de mayor tamaño se almacenan como archivos secuenciales en disco.

Si un índice es lo bastante pequeño como para guardarlo en la memoria principal, el tiempo de búsqueda para encontrar una entrada será breve. Sin embargo, si el índice es tan grande que se debe guardar en disco, buscar una entrada implicará leer varios bloques de disco. Para localizar una entrada en el archivo índice se puede realizar una búsqueda binaria, pero aun así ésta conlleva un gran coste. Si el índice ocupa  $b$  bloques, la búsqueda binaria tendrá que leer a lo sumo  $\lceil \log_2(b) \rceil$  bloques. ( $\lceil x \rceil$  denota el menor entero que es mayor o igual que  $x$ ; es decir, se redondea hacia abajo). Para el índice de 100 bloques, la búsqueda binaria necesitará leer siete bloques. En un disco en el que la lectura de un bloque tarda 30 milisegundos, la búsqueda empleará 210 milisegundos, que es bastante. Obsérvese que si se están usando bloques de desbordamiento, la búsqueda binaria no será posible. En ese caso, lo normal es una

**Figura 12.3** Índice disperso.



**Figura 12.4** Índice disperso de dos niveles.

búsqueda secuencial, y eso requiere leer  $b$  bloques, lo que podría consumir incluso más tiempo. Así, el proceso de buscar en un índice grande puede ser muy costoso.

Para resolver este problema el índice se trata como si fuese un archivo secuencial y se construye un índice disperso sobre el índice con agrupación, como se muestra en la Figura 12.4. Para localizar un registro se usa en primer lugar una búsqueda binaria sobre el índice más externo para buscar el registro con el mayor valor de la clave de búsqueda que sea menor o igual al valor deseado. El puntero apunta a un bloque en el índice más interno. Hay que examinar este bloque hasta encontrar el registro con el mayor valor de la clave que sea menor o igual que el valor deseado. El puntero de este registro apunta al bloque del archivo que contiene el registro buscado.

Usando los dos niveles de indexación y con el índice más externo en memoria principal hay que leer un único bloque índice, en vez de los siete que se leían con la búsqueda binaria. Si el archivo es extremadamente grande, incluso el índice exterior podría crecer demasiado para caber en la memoria principal. En este caso se podría crear todavía otro nivel más de indexación. De hecho, se podría repetir este proceso tantas veces como fuese necesario. Los índices con dos o más niveles se llaman índices **multinivel**. La búsqueda de registros usando un índice multinivel necesita claramente menos operaciones de E/S que las que se emplean en la búsqueda de registros con la búsqueda binaria. Cada nivel de índice se podría corresponder con una unidad del almacenamiento físico. Así, es posible disponer de índices para pistas, cilindros y discos.

Un diccionario normal es un ejemplo de un índice multinivel en el mundo ajeno a las bases de datos. La cabecera de cada página lista la primera palabra en el orden alfabético en esa página. Este índice es multinivel: las palabras en la parte superior de la página del índice del libro forman un índice disperso sobre los contenidos de las páginas del diccionario.

Los índices multinivel están estrechamente relacionados con la estructura de árbol, tales como los árboles binarios usados para la indexación en memoria. Esta relación se examinará posteriormente en el Apartado 12.3.

### 12.2.3 Actualización del índice

Sin importar el tipo de índice que se esté usando, los índices se deben actualizar siempre que se inserte o borre un registro del archivo. A continuación se describirán los algoritmos para actualizar índices de un solo nivel.

- **Inserción.** Primero se realiza una búsqueda usando el valor de la clave de búsqueda del registro a insertar. Las acciones que emprende el sistema a continuación dependen de si el índice es denso o disperso.

- Índices densos:

- 1. Si el valor de la clave de búsqueda no aparece en el índice, el sistema inserta en éste un registro índice con el valor de la clave de búsqueda en la posición adecuada.

- 2. En caso contrario se emprenden las siguientes acciones:

- a. Si el registro índice almacena punteros a todos los registros con el mismo valor de la clave de búsqueda, el sistema añade un puntero al nuevo registro en el registro índice.

- b. En caso contrario, el registro índice almacena un puntero sólo hacia el primer registro con el valor de la clave de búsqueda. El sistema sitúa el registro insertado después de los otros con los mismos valores de la clave de búsqueda.

- Índices dispersos: se asume que el índice almacena una entrada por cada bloque. Si el sistema crea un bloque nuevo, inserta el primer valor de la clave de búsqueda (en el orden de la clave de búsqueda) que aparezca en el nuevo bloque del índice. Por otra parte, si el nuevo registro tiene el menor valor de la clave de búsqueda en su bloque, el sistema actualiza la entrada del índice que apunta al bloque; si no, el sistema no realiza ningún cambio sobre el índice.

- **Borrado.** Para borrar un registro, primero se busca el índice a borrar. Las acciones que emprende el sistema a continuación dependen de si el índice es denso o disperso.

- Índices densos:

- 1. Si el registro borrado era el único registro con ese valor de la clave de búsqueda, el sistema borra el registro índice correspondiente del índice.

- 2. En caso contrario se emprenden las siguientes acciones:

- a. Si el registro índice almacena punteros a todos los registros con el mismo valor de la clave de búsqueda, el sistema borra del registro índice el puntero al registro borrado.

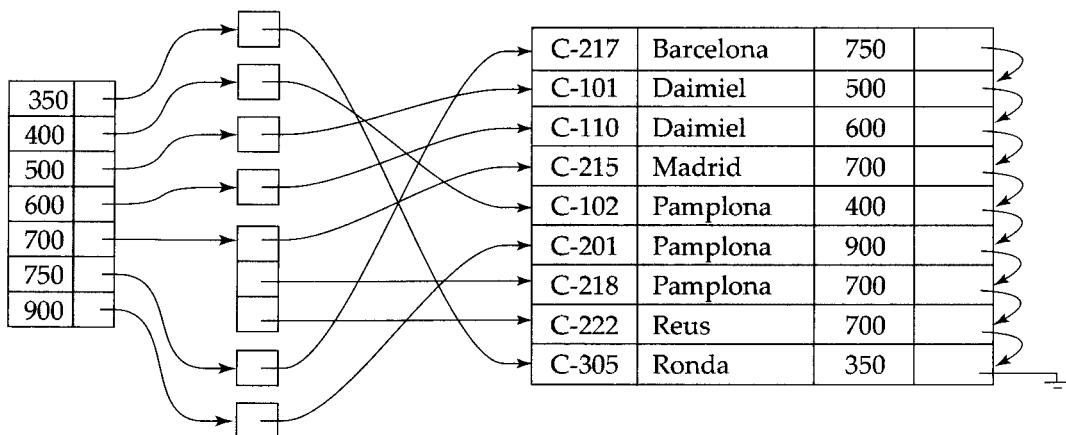
- b. En caso contrario, el registro índice almacena un puntero sólo al primer registro con el valor de la clave de búsqueda. En este caso, si el registro borrado era el primer registro con el valor de la clave de búsqueda, el sistema actualiza el registro índice para apuntar al siguiente registro.

- Índices dispersos:

- 1. Si el índice no contiene un registro índice con el valor de la clave de búsqueda del registro borrado, no hay que hacer nada.

- 2. En caso contrario se emprenden las siguientes acciones:

- a. Si el registro borrado era el único registro con la clave de búsqueda, el sistema reemplaza el registro índice correspondiente con un registro índice para el siguiente valor de la clave de búsqueda (en el orden de la clave de búsqueda). Si el siguiente valor de la clave de búsqueda ya tiene una entrada en el índice, se borra en lugar de reemplazarla.



**Figura 12.5** Índice secundario del archivo *cuenta*, con la clave no candidata *saldo*.

- b. En caso contrario, si el registro índice para el valor de la clave de búsqueda apunta al registro a borrar, el sistema actualiza el registro índice para que apunte al siguiente registro con el mismo valor de la clave de búsqueda.

Los algoritmos de inserción y borrado para los índices multinivel se extienden de manera sencilla a partir del esquema descrito anteriormente. Al borrar o al insertar se actualiza el índice de nivel más bajo como se describió anteriormente. Por lo que respecta al índice del segundo nivel, el índice de nivel más bajo es simplemente un archivo de registros—así, si hay algún cambio en el índice de nivel más bajo, se tendrá que actualizar el índice del segundo nivel como ya se describió. La misma técnica se aplica al resto de niveles del índice, si los hubiera.

## 12.2.4 Índices secundarios

Los índices secundarios deben ser densos, con una entrada en el índice por cada valor de la clave de búsqueda, y un puntero a cada registro del archivo. Un índice con agrupación puede ser disperso, almacenando sólo algunos de los valores de la clave de búsqueda, ya que siempre es posible encontrar registros con valores de la clave de búsqueda intermedios mediante un acceso secuencial a parte del archivo, como se describió antes. Si un índice secundario almacena sólo algunos de los valores de la clave de búsqueda, los registros con los valores de la clave de búsqueda intermedios pueden estar en cualquier lugar del archivo y, en general, no se pueden encontrar sin explorar el archivo completo.

Un índice secundario sobre una clave candidata es como un índice denso con agrupación, excepto en que los registros apuntados por los sucesivos valores del índice no están almacenados secuencialmente. Generalmente los índices secundarios están estructurados de manera diferente a como lo están los índices con agrupación. Si la clave de búsqueda de un índice con agrupación no es una clave candidata, es suficiente si el valor de cada entrada en el índice apunta al primer registro con ese valor en la clave de búsqueda, ya que los otros registros podrían ser alcanzados por una búsqueda secuencial del archivo.

En cambio, si la clave de búsqueda de un índice secundario no es una clave candidata, no sería suficiente apuntar sólo al primer registro de cada valor de la clave. El resto de registros con el mismo valor de la clave de búsqueda podría estar en cualquier otro sitio del archivo, ya que los registros están ordenados según la clave de búsqueda del índice con agrupación, en vez de la clave de búsqueda del índice secundario. Por tanto, un índice secundario debe contener punteros a todos los registros.

Se puede usar un nivel adicional de referencia para implementar los índices secundarios sobre claves de búsqueda que no sean claves candidatas. Los punteros en estos índices secundarios no apuntan directamente al archivo. En vez de eso, cada puntero apunta a un cajón que contiene punteros al archivo. En la Figura 12.5 se muestra la estructura del archivo *cuenta*, con un índice secundario que usa un nivel de referencia adicional, y teniendo como clave de búsqueda el *saldo*.

Siguiendo el orden de un índice primario, una búsqueda secuencial es eficiente porque los registros del archivo están guardados físicamente de la misma manera a como está ordenado el índice. Sin embar-

go, no se puede (salvo en casos excepcionales) almacenar el archivo ordenado físicamente por el orden de la clave de búsqueda del índice con agrupación y la clave de búsqueda del índice secundario. Ya que el orden de la clave secundaria y el orden físico difieren, si se intenta examinar el archivo secuencialmente por el orden de la clave secundaria, es muy probable que la lectura de cada bloque suponga la lectura de un nuevo bloque del disco, lo cual es muy lento.

El procedimiento ya descrito para borrar e insertar se puede aplicar también a los índices secundarios; las acciones a emprender son las descritas para los índices densos que almacenan un puntero a cada registro del archivo. Si un archivo tiene varios índices, siempre que se modifique el archivo, se debe actualizar *cada* uno de ellos.

Los índices secundarios mejoran el rendimiento de las consultas que usan claves que no son la de búsqueda del índice con agrupación. Sin embargo, implican un tiempo adicional importante al modificar la base de datos. El diseñador de la base de datos debe decidir qué índices secundarios son deseables, según una estimación sobre las frecuencias de las consultas y de las modificaciones.

## 12.3 Archivos de índices de árbol B<sup>+</sup>

El inconveniente principal de la organización de un archivo secuencial indexado reside en que el rendimiento, tanto para buscar en el índice como para buscar secuencialmente a través de los datos, se degrada según crece el archivo. Aunque esta degradación se puede remediar reorganizando el archivo, no es deseable hacerlo frecuentemente.

La estructura de índice de **árbol B<sup>+</sup>** es la más extendida de las estructuras de índices que mantienen su eficiencia a pesar de la inserción y borrado de datos. Un índice de árbol B<sup>+</sup> toma la forma de un **árbol equilibrado** donde los caminos de la raíz a cada hoja del árbol son de la misma longitud. Cada nodo que no sea hoja tiene entre  $[n/2]$  y  $n$  hijos, donde  $n$  es fijo para cada árbol concreto.

Se verá que la estructura de árbol B<sup>+</sup> implica una degradación del rendimiento al insertar y al borrar, además de un espacio extra. Este tiempo adicional es aceptable incluso en archivos con altas frecuencias de modificación, ya que se evita el coste de reorganizar el archivo. Además, puesto que los nodos podrían estar a lo sumo medio llenos (si tienen el mínimo número de hijos) se desperdicia algo de espacio. Este gasto de espacio adicional también es aceptable dados los beneficios en el rendimiento aportados por las estructura de árbol B<sup>+</sup>.

### 12.3.1 Estructura de árbol B<sup>+</sup>

Un índice de árbol B<sup>+</sup> es un índice multinivel pero con una estructura que difiere del índice multinivel de un archivo secuencial. En la Figura 12.6 se muestra un nodo típico de un árbol B<sup>+</sup>. Puede contener hasta  $n - 1$  claves de búsqueda  $K_1, K_2, \dots, K_{n-1}$  y  $n$  punteros  $P_1, P_2, \dots, P_n$ . Los valores de la clave de búsqueda de un nodo se mantienen ordenados; así, si  $i < j$ , entonces  $K_i < K_j$ .

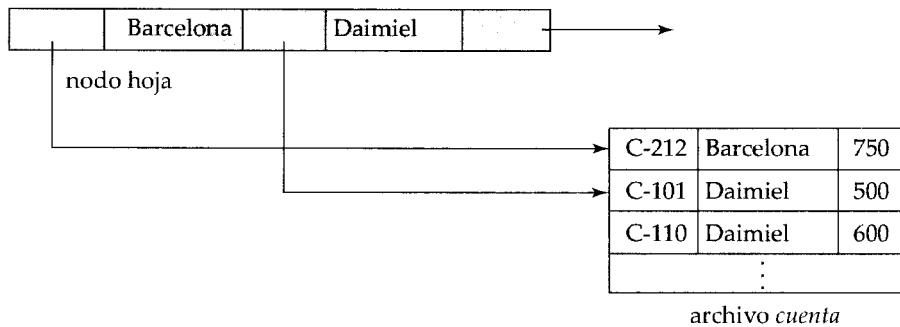
Considérese primero la estructura de los nodos hoja. Para  $i = 1, 2, \dots, n - 1$ , el puntero  $P_i$  apunta, o bien a un registro del archivo con valor de la clave de búsqueda  $K_i$ , o bien a un cajón de punteros, cada uno de los cuales apunta a un registro del archivo con valor de la clave de búsqueda  $K_i$ . La estructura cajón se usa solamente si la clave de búsqueda no es una clave candidata y si el archivo no está ordenado según la clave de búsqueda (en el Apartado 12.5.3 se estudia la forma de evitar la creación de cajones haciendo que parezca que la clave de búsqueda sea única). El puntero  $P_n$  tiene un propósito especial que se estudiará más adelante.

En la Figura 12.7 se muestra un nodo hoja en el árbol B<sup>+</sup> del archivo *cuenta*, donde  $n$  vale tres y la clave de búsqueda es *nombre\_sucursal*. Obsérvese que, como el archivo cuenta está ordenado por *nombre\_sucursal*, los punteros en el nodo hoja apuntan directamente al archivo.

Ahora que se ha visto la estructura de un nodo hoja, se mostrará cómo los valores de la clave de búsqueda se asignan a nodos concretos. Cada hoja puede guardar hasta  $n - 1$  valores. Está permitido

|       |       |       |         |           |           |       |
|-------|-------|-------|---------|-----------|-----------|-------|
| $P_1$ | $K_1$ | $P_2$ | $\dots$ | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|-------|-------|-------|---------|-----------|-----------|-------|

Figura 12.6 Nodo típico de un árbol B<sup>+</sup>.



**Figura 12.7** Nodo hoja para el índice del árbol B<sup>+</sup> de *cuenta* ( $n = 3$ ).

que los nodos hojas contengan al menos  $\lceil(n - 1)/2\rceil$  valores. Los rangos de los valores en cada hoja no se solapan. Así, si  $L_i$  y  $L_j$  son nodos hoja e  $i < j$ , entonces cada valor de la clave de búsqueda en  $L_i$  es menor que cada valor de la clave en  $L_j$ . Si el índice de árbol B<sup>+</sup> es un índice denso, cada valor de la clave de búsqueda debe aparecer en algún nodo hoja.

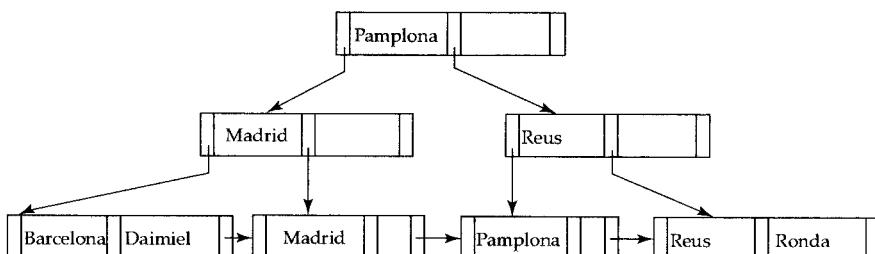
Ahora se puede explicar el uso del puntero  $P_n$ . Dado que existe un orden lineal en las hojas basado en los valores de la clave de búsqueda que contienen, se usa  $P_n$  para encadenar juntos los nodos hojas en el orden de la clave de búsqueda. Esta ordenación permite un procesamiento secuencial eficaz del archivo.

Los nodos internos del árbol B<sup>+</sup> forman un índice multinivel (disperso) sobre los nodos hoja. La estructura de los nodos internos es la misma que la de los nodos hoja, excepto que todos los punteros son punteros a nodos del árbol. Un nodo interno podría guardar hasta  $n$  punteros y *debe* guardar al menos  $\lceil n/2 \rceil$  punteros. El número de punteros de un nodo se llama *grado de salida* del nodo.

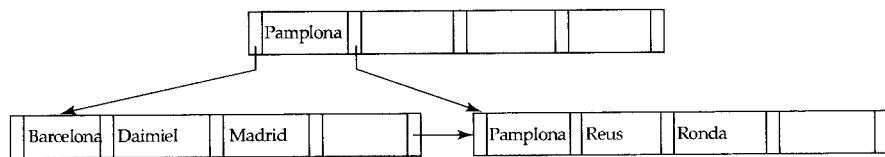
Considérese un nodo que contiene  $m$  punteros. Para  $i = 2, 3, \dots, m - 1$ , el puntero  $P_i$  apunta al subárbol que contiene los valores de la clave de búsqueda menores que  $K_i$  y mayor o igual que  $K_{i-1}$ . El puntero  $P_m$  apunta a la parte del subárbol que contiene los valores de la clave mayores o iguales que  $K_{m-1}$ , y el puntero  $P_1$  apunta a la parte del subárbol que contiene los valores de la clave menores que  $K_1$ .

A diferencia de otros nodos internos, el nodo raíz puede tener menos de  $\lceil n/2 \rceil$ ; sin embargo, debe tener al menos dos punteros, salvo que el árbol tenga un solo nodo. Siempre es posible construir un árbol B<sup>+</sup>, para cualquier  $n$ , que satisfaga los requisitos anteriores. En la Figura 12.8 se muestra un árbol B<sup>+</sup> completo para el archivo *cuenta* ( $n = 3$ ). Por simplicidad se omiten los punteros al propio archivo y los punteros nulos. Como ejemplo de un árbol B<sup>+</sup> en el cual la raíz debe tener menos de  $\lceil n/2 \rceil$  valores, en la Figura 12.9 se muestra un árbol B<sup>+</sup> para el archivo *cuenta* con  $n = 5$ .

En todos los ejemplos mostrados de árboles B<sup>+</sup>, éstos están equilibrados. Es decir, la longitud de cada camino desde la raíz a cada nodo hoja es la misma. Esta propiedad es un requisito de los árboles B<sup>+</sup>. De hecho, la “B” de árbol B<sup>+</sup> proviene del inglés “balanced” (equilibrado). Es esta propiedad de equilibrio de los árboles B<sup>+</sup> la que asegura un buen rendimiento para las búsquedas, inserciones y borrados.



**Figura 12.8** Árbol B<sup>+</sup> para el archivo *cuenta* ( $n = 3$ ).



**Figura 12.9** Árbol B<sup>+</sup> para el archivo *cuenta* ( $n = 5$ ).

### 12.3.2 Consultas con árboles B<sup>+</sup>

Considérese ahora cómo procesar consultas usando árboles B<sup>+</sup>. Supóngase que se desea encontrar todos los registros cuyo valor de la clave de búsqueda sea  $V$ . La Figura 12.10 muestra el pseudocódigo para hacerlo. Primero se examina el nodo raíz para buscar el menor valor de la clave de búsqueda mayor que  $V$ . Supóngase que este valor de la clave de búsqueda es  $K_i$ . Después se sigue el puntero  $P_i$  hasta otro nodo. Si no se encuentra ese valor, entonces  $k \geq K_{m-1}$ , donde  $m$  es el número de punteros del nodo. Es este caso se sigue  $P_m$  hasta otro nodo. En el nodo alcanzado anteriormente se busca de nuevo el menor valor de la clave de búsqueda que es mayor que  $V$  para seguir el puntero correspondiente. Finalmente se alcanza un nodo hoja. En este nodo hoja, si se encuentra que el valor  $K_i$  es igual a  $V$ , entonces el puntero  $P_i$  nos ha conducido al registro o cajón deseado. Si no se encuentra el valor  $V$  en el nodo hoja, no existe ningún registro con el valor clave  $V$ .

De esta manera, para procesar una consulta, se tiene que recorrer un camino en el árbol desde la raíz hasta algún nodo hoja. Si hay  $K$  valores de la clave de búsqueda en el archivo, este camino no será más largo que  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ .

En la práctica sólo se accede a unos cuantos nodos. Generalmente un nodo se construye para tener el mismo tamaño que un bloque de disco, el cual ocupa normalmente 4 KB. Con una clave de búsqueda del tamaño de 12 bytes y un tamaño del puntero a disco de 8 bytes,  $n$  está alrededor de 200. Incluso con una estimación más conservadora de 32 bytes para el tamaño de la clave de búsqueda,  $n$  está entorno a 100. Con  $n = 100$ , si se tienen un millón de valores de la clave de búsqueda en el archivo, una búsqueda necesita solamente  $\lceil \log_{50}(1.000.000) \rceil = 4$  accesos a nodos. Por tanto, se necesitan leer a lo sumo cuatro bloques del disco para realizar la búsqueda. Normalmente, se accede mucho al nodo raíz del árbol y, por ello, suele guardarse en una memoria intermedia; por tanto, solamente hace falta leer, como máximo, tres bloques del disco.

Una diferencia importante entre las estructuras de árbol B<sup>+</sup> y los árboles en memoria, tales como los árboles binarios, está en el tamaño de los nodos y, por tanto, la altura del árbol. En los árboles binarios, todos los nodos son pequeños y tienen, a lo sumo, dos punteros. En los árboles B<sup>+</sup>, cada nodo es grande —normalmente un bloque del disco— y puede tener gran número de punteros. Así, los árboles B<sup>+</sup> tienden a ser bajos y anchos, en lugar de los altos y estrechos árboles binarios. En un árbol equilibrado, el camino de una búsqueda puede tener una longitud de  $\lceil \log_2(K) \rceil$ , donde  $K$  es el número de valores

```

procedure buscar(valor V)
 C = nodo raíz
 while C no sea un nodo raíz begin
 K_i = mínimo valor de la clave de búsqueda, si lo hay, mayor que V
 if no hay tal valor then begin
 m = número de punteros en el nodo
 C = nodo al que apunta P_m
 end
 else C = el nodo al que apunta P_i
 end
 if hay un valor de la clave K_i en C tal que $K_i = V$
 then el puntero P_i conduce al registro o cajón deseado
 else no existe ningún registro con el valor de la clave k

```

**Figura 12.10** Consulta con un árbol B<sup>+</sup>.

de la clave de búsqueda. Con  $K = 1.000.000$ , como en el ejemplo anterior, un árbol binario equilibrado necesita alrededor de 20 accesos a nodos. Si cada nodo estuviera en un bloque del disco distinto, serían necesarias 20 lecturas a bloques para procesar la búsqueda, en contraste con las cuatro lecturas del árbol B<sup>+</sup>.

### 12.3.3 Actualizaciones en árboles B<sup>+</sup>

El borrado y la inserción son operaciones más complicadas que las búsquedas, ya que podría ser necesario **dividir** un nodo que fuese demasiado grande como resultado de una inserción, o **fusionar** nodos si un nodo se volviera demasiado pequeño (menor que  $\lceil n/2 \rceil$  punteros). Además, cuando se divide un nodo o se fusionan un par de ellos, se debe asegurar que el equilibrio del árbol se mantenga. Para presentar la idea que hay detrás del borrado y la inserción en un árbol B<sup>+</sup>, se asumirá que los nodos nunca serán demasiado grandes ni demasiado pequeños. Bajo esta suposición, el borrado y la inserción se realizan como se indica a continuación.

- **Inserción.** Usando la misma técnica que para buscar, se busca un nodo hoja donde tendría que aparecer el valor de la clave de búsqueda. Si el valor de la clave de búsqueda ya aparece en el nodo hoja, se inserta un nuevo registro en el archivo y, si es necesario, un puntero al cajón. Si el valor de la clave de búsqueda no aparece, se inserta el valor en el nodo hoja de tal manera que las claves de búsqueda permanezcan ordenadas. Luego se inserta el nuevo registro en el archivo y, si es necesario, se crea un nuevo cajón con el puntero apropiado.
- **Borrado.** Usando la misma técnica que para buscar, se busca el registro a borrar y se elimina del archivo. Si no existe un cajón asociado con el valor de la clave de búsqueda o si el cajón se queda vacío como resultado del borrado, se borra el valor de la clave de búsqueda del nodo hoja.

A continuación se considera un ejemplo en el que se tiene que dividir un nodo. Por ejemplo, insertar un registro en el árbol B<sup>+</sup> de la Figura 12.8, cuyo valor de *nombre\_sucursal* es “Cádiz”. Usando el algoritmo de búsqueda, “Cádiz” debería aparecer en el nodo que incluye “Barcelona” y “Daimiel”. No hay sitio para insertar el valor de la clave de búsqueda “Cádiz”. Por tanto, se *divide* el nodo en otros dos nodos. En la Figura 12.11 se muestran los nodos hoja que resultan de insertar “Cádiz” y de dividir el nodo que incluía “Barcelona” y “Daimiel”. En general, si existen  $n$  valores de la clave de búsqueda (los  $n - 1$  valores del nodo hoja más el valor que se inserta), se asignará  $\lceil n/2 \rceil$  al nodo existente y el resto de valores en el nuevo nodo.

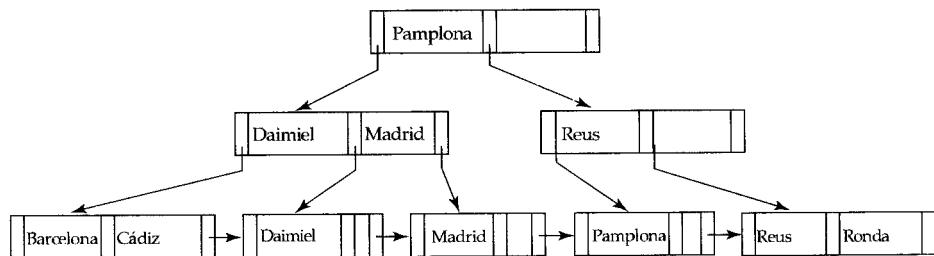
Después de dividir el nodo hoja hay que insertar el nuevo nodo hoja en el árbol B<sup>+</sup>. En el nuevo nodo del ejemplo, “Daimiel” es el valor más pequeño de la clave de búsqueda. Después hay que insertar este valor de la clave de búsqueda en el padre del nodo hoja dividido. En el árbol B<sup>+</sup> de la Figura 12.12 se muestra el resultado de la inserción. El valor “Daimiel” de la clave de búsqueda se ha insertado en el padre. Ha sido posible llevar a cabo esta inserción porque había sitio para añadir un valor de la clave de búsqueda. En caso de no haber sitio se dividiría el padre. En el peor de los casos, todos los nodos en el camino hacia la raíz se tendrían que dividir. Si la propia raíz se tuviera que dividir, el árbol hubiera sido más profundo.

La técnica general para la inserción en un árbol B<sup>+</sup> es determinar el nodo hoja  $h$  en el cual realizar la inserción. Si es necesario dividir, se inserta el nuevo nodo dentro del padre del nodo  $h$ . Si esta inserción produce otra división, se procedería recursivamente o bien hasta que una inserción no produzca otra división o bien hasta crear una nueva raíz.

En la Figura 12.13 se bosqueja el algoritmo de inserción en pseudocódigo. El procedimiento inserta un par valor de la clave y puntero en el índice usando los procedimientos insertar\_en\_hoja e inser-



**Figura 12.11** División del nodo hoja tras la inserción de “Cádiz”.



**Figura 12.12** Inserción de “Cádiz” en el árbol B<sup>+</sup> de la Figura 12.8.

tar\_en\_padre. En el pseudocódigo  $L, N, P$  y  $T$  denotan punteros a nodos y  $L$  se usa para denotar un nodo hoja.  $L.K_i$  y  $L.P_i$  denotan el  $i$ -ésimo valor y el  $i$ -ésimo puntero en el nodo  $L$ , respectivamente.  $T.K_i$  y  $T.P_i$  se usan de forma análoga. El pseudocódigo también emplea la función  $padre(N)$  para encontrar el padre del nodo  $N$ . Se puede obtener una lista de los nodos en el camino de la raíz a la hoja al buscar inicialmente el nodo hoja, y se puede usar después para encontrar eficazmente el padre de cualquier nodo del camino.

El procedimiento insertar\_en\_padre tiene como parámetros  $N, K'$  y  $N'$ , donde el nodo  $N$  se ha dividido en  $N$  y  $N'$ , siendo  $K'$  el valor mínimo en  $N'$ . El procedimiento modifica el padre de  $N$  para registrar la división. Los procedimientos insertar\_en\_hoja e insertar\_en\_padre usan el área temporal de memoria  $T$  para almacenar los contenidos del nodo que se está dividiendo. Los procedimientos se pueden modificar para que copien directamente los datos del nodo que se divide en el nodo que se crea, reduciendo el tiempo necesario para la copia. Sin embargo, el uso del espacio temporal  $T$  simplifica los procedimientos.

A continuación se consideran los borrados que provocan que el árbol se quede con muy pocos punteros. Primero se borra “Daimiel” del árbol B<sup>+</sup> de la Figura 12.12. Para ello se localiza la entrada “Daimiel” usando el algoritmo de búsqueda. Cuando se borra la entrada “Daimiel” de su nodo hoja, la hoja se queda vacía. Ya que en el ejemplo  $n = 3$  y  $0 < \lceil(n - 1)/2\rceil$ , este nodo se debe borrar del árbol B<sup>+</sup>. Para borrar un nodo hoja se tiene que borrar el puntero que le llega desde su padre. En el ejemplo, este borrado deja al nodo padre, el cual contenía tres punteros, con sólo dos punteros. Ya que  $2 \geq \lceil n/2 \rceil$ , el nodo es todavía lo suficientemente grande y la operación de borrado se completa. El árbol B<sup>+</sup> resultante se muestra en la Figura 12.14.

Cuando un borrado se hace sobre el padre de un nodo hoja, el propio nodo padre podría quedar demasiado pequeño. Esto es exactamente lo que ocurre si se borra “Pamplona” del árbol B<sup>+</sup> de la Figura 12.14. El borrado de la entrada Pamplona provoca que un nodo hoja se quede vacío. Cuando se borra el puntero a este nodo en su padre, éste sólo se queda con un puntero. Así,  $n = 3$ ,  $\lceil n/2 \rceil = 2$  y queda tan sólo un puntero, que es demasiado poco. Sin embargo, ya que el nodo padre contiene información útil, no basta simplemente con borrarlo. En vez de eso, se busca el nodo hermano (el nodo interno que contiene al menos una clave de búsqueda, Madrid). Este nodo hermano dispone de sitio para colocar la información contenida en el nodo que quedó demasiado pequeño, así que se fusionan estos nodos, de tal manera que el nodo hermano ahora contiene las claves “Madrid” y “Reus”. El otro nodo (el nodo que contenía solamente la clave de búsqueda “Reus”) ahora contiene información redundante y se puede borrar desde su padre (el cual resulta ser la raíz del ejemplo). En la Figura 12.15 se muestra el resultado. Hay que observar que la raíz tiene solamente un puntero hijo como resultado del borrado, así que éste se borra y el hijo solitario se convierte en la nueva raíz. De esta manera la profundidad del árbol ha disminuido en una unidad.

No siempre es posible fusionar nodos. Como ejemplo se borrará “Pamplona” del árbol B<sup>+</sup> de la Figura 12.11. En este ejemplo, la entrada “Daimiel” es todavía parte del árbol. Una vez más, el nodo hoja que contiene “Pamplona” se queda vacío. El padre del nodo hoja se queda también demasiado pequeño (únicamente con un puntero). De cualquier modo, en este ejemplo, el nodo hermano contiene ya el máximo número de punteros: tres. Así, no puede acomodar a un puntero adicional. La solución en este caso es **redistribuir** los punteros de tal manera que cada hermano tenga dos punteros. El resultado se muestra en la Figura 12.16. Obsérvese que la redistribución de los valores necesita de un cambio en el valor de la clave de búsqueda en el padre de los dos hermanos.

```

procedure insertar(valor K, puntero P)
 hallar el nodo hoja L que debe contener el valor de la clave K
 if (L tiene menos de n – 1 valores de la clave)
 then insertar_en_hoja (L, K, P)
 else begin /* L ya tiene n – 1 valores de la clave, dividirlo */
 Crear nodo L'
 Copiar L.P1 ... L.Kn-1 a un bloque de memoria T que pueda
 almacenar n pares (puntero, valor de la clave)
 insertar_en_hoja (T, K, P)
 L'.Pn = L.Pn; L.Pn = L'
 Borrar desde L.P1 hasta L.Kn-1 de L
 Copiar desde T.P1 hasta T.K⌈n/2⌉ de T a L comenzando en L.P1
 Copiar desde T.P⌈n/2⌉+1 hasta T.Kn de T a L' comenzando en L'.P1
 K' el valor mínimo de la clave en L'
 insertar_en_padre(L, K', L')
 end

procedure insertar_en_hoja (nodo L, valor K, puntero P)
 if K es menor que L.K1
 then insertar P, K en L justo antes de L.P1
 else begin
 Ki el mayor valor de L que sea menor que K
 insertar P, K en L justo después de T.Ki
 end

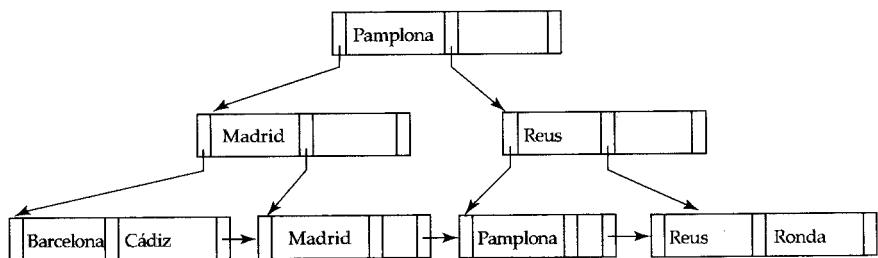
procedure insertar_en_padre(nodo N, valor K', nodo N')
 if N es la raíz del árbol
 • then begin
 crear un nuevo nodo R que contenga N, K', N' /* N y N' son punteros */
 hacer de R la raíz del árbol
 return
 end
 P = padre (N)
 if (P tiene menos de n punteros)
 then insertar (K', N') en P justo después de N
 else begin /* Dividir P */
 Copiar P a un bloque de memoria T que pueda almacenar P y (K', N')
 Insertar (K', N') en T justo después de N
 Borrar todas las entradas de P; Crear el nodo P'
 Copiar T.P1 ... T.P⌈n/2⌉ en P
 K'' = T.K⌈n/2⌉
 Copiar T.P⌈n/2⌉+1 ... T.Pn+1 en P'
 insertar_en_padre(P, K'', P')
 end

```

**Figura 12.13** Inserción de una entrada en un árbol B<sup>+</sup>.

En general, para borrar un valor en un árbol B<sup>+</sup> se realiza una búsqueda según el valor y se borra. Si el nodo es demasiado pequeño, se borra desde su padre. Este borrado se realiza como una aplicación recursiva del algoritmo de borrado hasta que se alcanza la raíz, un nodo padre queda lleno de manera adecuada después de borrar, o hasta aplicar una redistribución.

En la Figura 12.17 se describe el pseudocódigo para el borrado en un árbol B<sup>+</sup>. El procedimiento intercambiar\_variables(*N, N'*) simplemente cambia de lugar los valores (punteros) de las variables *N* y *N'*; este cambio no afecta al árbol en sí mismo. El pseudocódigo utiliza la condición “muy pocos va-



**Figura 12.14** Borrado de “Daimiel” del ´rbol B<sup>+</sup> de la Figura 12.12.

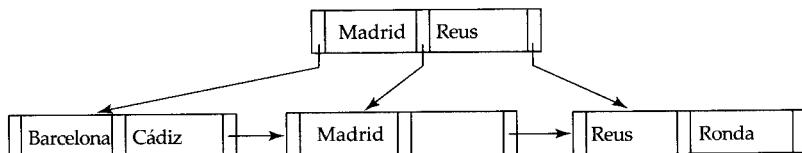
lores/punteros”. Para nodos internos, este criterio quiere decir: menos que  $\lceil n/2 \rceil$  punteros; para nodos hoja, quiere decir: menos que  $\lceil (n - 1)/2 \rceil$  valores. El pseudocodo realiza la redistribucion tomando prestada una sola entrada desde un nodo adyacente. Tambien se puede redistribuir mediante la distribucion equitativa de entradas entre dos nodos. El pseudocodo hace referencia al borrado de una entrada  $(K, P)$  desde un nodo. En el caso de los nodos hoja, el puntero a una entrada realmente precede al valor de la clave; asi, el puntero  $P$  precede al valor de la clave  $K$ . Para nodos internos,  $P$  sigue al valor de la clave  $K$ .

Es interesante señalar que, como resultado de un borrado, puede que haya valores de la clave de nodos internos de un ´rbol B<sup>+</sup> que no esten en ninguna hoja del ´rbol.

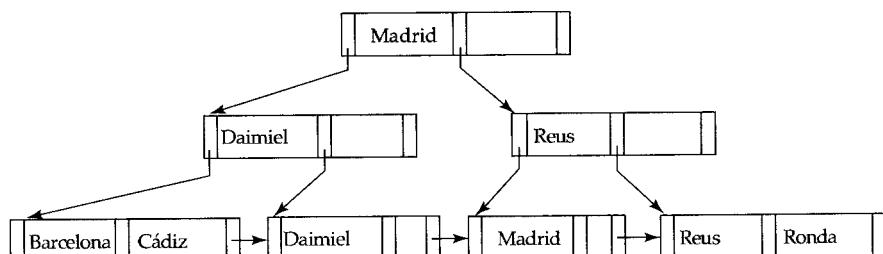
Aunque las operaciones insercion y borrado en ´rboles B<sup>+</sup> son complicadas, requieren relativamente pocas operaciones E/S, lo que es un beneficio importante dado su coste. Se puede demostrar que el numero de operaciones E/S necesarias para una insercion o borrado es, en el peor de los casos, proporcional a  $\log_{\lceil n/2 \rceil}(K)$ , donde  $n$  es el numero maximo de punteros en un nodo y  $K$  es el numero de valores de la clave de bqueda. En otras palabras, el coste de las operaciones insercion y borrado es proporcional a la altura del ´rbol B<sup>+</sup> y es, por tanto, bajo. Debido a la velocidad de las operaciones en los ´rboles B<sup>+</sup>, estas estructuras de ´ndice se usan frecuentemente al implementar las bases de datos.

### 12.3.4 Organizacion de archivos con ´rboles B<sup>+</sup>

Como se menciono en el Apartado 12.3, el inconveniente de la organizacion de archivos secuenciales de ´ndices es la degradacion del rendimiento segun crece el archivo: con el crecimiento, un porcentaje mayor de registros ´ndice y registros reales se desaprueban y se almacenan en bloques de desbordamiento. La degradacion de las bquedas en el ´ndice se resuelve mediante el uso de ´ndices de ´rbol B<sup>+</sup> en el archivo. Tambien se soluciona el problema de la degradacion al almacenar los registros reales



**Figura 12.15** Borrado de “Pamplona” del ´rbol B<sup>+</sup> de la Figura 12.14.



**Figura 12.16** Borrado de “Pamplona” del ´rbol B<sup>+</sup> de la Figura 12.12.

```

procedure borrar(valor K, puntero P)
 hallar el nodo hoja L que contiene (K, P)
 borrar_entrada(L, K, P)

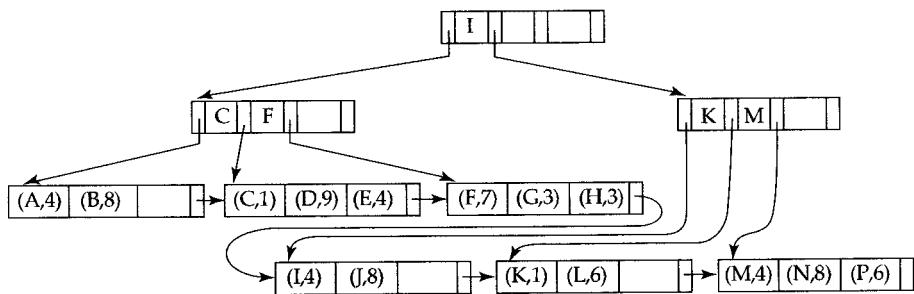
procedure borrar_entrada(nodo N, valor K, puntero P)
 borrar (K, P) de N
 if (N es la raíz and a N sólo le queda un hijo)
 then hacer del hijo de N la nueva raíz del árbol y borrar N
 else if (N tiene muy pocos valores/punteros) then begin
 sea N' el hijo anterior o siguiente de padre(N)
 sea K' el valor entre los punteros N y N' en padre(N)
 if (las entradas en N y N' caben en un solo nodo)
 then begin /* Fusionar los nodos */
 if (N es predecesor de N') then intercambiar_variables(N, N')
 if (N no es una hoja)
 then concatenar K' y todos los punteros y valores en N a N'
 else concatenar todos los pares (Ki, Pi) en N a N'; sea N'.Pn = N.Pn
 borrar_entrada(padre(N), K', N); borrar el nodo N
 end
 else begin /* Redistribución: tomar prestada una entrada de N' */
 if (N' es predecesor de N) then begin
 if (N es un nodo interno) then begin
 sea m tal que N'.Pm es el último puntero en N'
 suprimir (N'.Km-1, N'.Pm) de N'
 insertar (N'.Pm, K') como primer puntero y valor en N,
 desplazando otros punteros y valores a la derecha
 sustituir K' en padre(N) por N'.Km-1
 end
 else begin
 sea m tal que (N'.Pm, N'.Km) es el último par puntero/valor
 en N'
 suprimir (N'.Pm, N'.Km) de N'
 insertar (N'.Pm, N'.Km) como primer puntero y valor en N,
 desplazando otros punteros y valores a la derecha
 sustituir K' en padre(N) por N'.Km
 end
 end
 else ... simétrico al caso then ...
 end
end

```

**Figura 12.17** Borrado de una entrada de un árbol B<sup>+</sup>.

utilizando el nivel de hoja del árbol B<sup>+</sup> para almacenar los registros reales en los bloques. En estas estructuras, la estructura del árbol B<sup>+</sup> se usa no sólo como un índice, sino también como un organizador de los registros dentro del archivo. En la **organización de archivo con árboles B<sup>+</sup>**, los nodos hoja del árbol almacenan registros, en lugar de almacenar punteros a registros. En la Figura 12.18 se muestra un ejemplo de la organización de un archivo con un árbol B<sup>+</sup>. Ya que los registros son normalmente más grandes que los punteros, el número máximo de registros que se pueden almacenar en un nodo hoja es menor que el número de punteros en un nodo interno. Sin embargo, todavía se requiere que los nodos hoja estén llenos al menos hasta la mitad.

La inserción y el borrado de registros en las organizaciones de archivos con árboles B<sup>+</sup> se tratan del mismo modo que la inserción y el borrado de entradas en los índices de árboles B<sup>+</sup>. Cuando se inserta un registro con un valor de clave *v*, se localiza el bloque que debería contener ese registro mediante la



**Figura 12.18** Organización de archivos con árboles B<sup>+</sup>.

búsqueda en el árbol B<sup>+</sup> de la mayor clave que sea  $\leq v$ . Si el bloque localizado tiene bastante espacio libre para el registro, se almacena el registro en el bloque. De no ser así, como en una inserción en un árbol B<sup>+</sup>, se divide el bloque en dos y se redistribuyen sus registros (en el orden de la clave del árbol B<sup>+</sup>) creando espacio para el nuevo registro. Esta división se propaga hacia arriba en el árbol B<sup>+</sup> de la manera usual. Cuando se borra un registro, primero se elimina del bloque que lo contiene. Si como resultado un bloque B llega a ocupar menos que la mitad, los registros en B se redistribuyen con los registros en un bloque B' adyacente. Si se asume que los registros son de tamaño fijo, cada bloque contendrá por lo menos la mitad de los registros que pueda contener como máximo. Los nodos internos del árbol B<sup>+</sup> se actualizan por tanto de la manera habitual.

Cuando un árbol B<sup>+</sup> se utiliza para la organización de un archivo, la utilización del espacio es particularmente importante, ya que el espacio ocupado por los registros es mucho mayor que el espacio ocupado por las claves y punteros. Se puede mejorar la utilización del espacio en un árbol B<sup>+</sup> implicando a más nodos hermanos en la redistribución durante las divisiones y fusiones. La técnica es aplicable a los nodos hoja y nodos internos y funciona como sigue.

Durante la inserción, si un nodo está lleno se intenta redistribuir algunas de sus entradas en uno de los nodos adyacentes para hacer sitio a una nueva entrada. Si este intento falla porque los nodos adyacentes están llenos, se divide el nodo y las entradas entre uno de los nodos adyacentes y los dos nodos que se obtienen al dividir el nodo original. Puesto que los tres nodos juntos contienen un registro más que puede encajar en dos nodos, cada nodo estará lleno aproximadamente hasta sus dos terceras partes. Para ser más precisos, cada nodo tendrá por lo menos  $\lfloor 2n/3 \rfloor$  entradas, donde n es el número máximo de entradas que puede tener un nodo ( $\lfloor x \rfloor$  denota el mayor entero menor o igual que x; es decir, se elimina la parte fraccionaria si la hay).

Durante el borrado de un registro, si la ocupación de un nodo está por debajo de  $\lfloor 2n/3 \rfloor$ , se intentará tomar prestada una entrada desde uno de sus nodos hermanos. Si ambos nodos hermanos tienen  $\lfloor 2n/3 \rfloor$  registros, en lugar de tomar prestada una entrada, se redistribuyen las entradas en el nodo y en los dos nodos hermanos uniformemente entre dos de los nodos y se borra el tercer nodo. Se puede usar este enfoque porque el número total de entradas es  $3\lfloor 2n/3 \rfloor - 1$ , lo cual es menor que  $2n$ . Utilizando tres nodos adyacentes para la redistribución se puede garantizar que cada nodo tenga  $\lfloor 3n/4 \rfloor$  entradas. En general, si hay m nodos ( $m - 1$  hermanos) implicados en la redistribución se puede garantizar que cada nodo contenga, al menos,  $\lfloor (m - 1)n/m \rfloor$  entradas. Sin embargo, el coste de la actualización se vuelve mayor según haya más nodos hermanos involucrados en la redistribución.

Obsérvese que en un índice de árbol B<sup>+</sup> los nodos hoja adyacentes en el árbol se puede ubicar en diferentes lugares del disco. Cuando se crea un índice sobre un conjunto de registros es posible asignar bloques contiguos en disco para nodos hoja que también lo sean en el árbol. Así, una exploración secuencial de los nodos hoja correspondería a una exploración secuencial en el disco. Como se dan inserciones y borrados en el árbol, se pierde la secuencialidad, y las esperas a disco en el acceso secuencial se incrementan. Sería necesario una reorganización del índice para mantener la secuencialidad.

Las organizaciones de archivo del tipo árbol B<sup>+</sup> se pueden usar para almacenar grandes objetos, como los tipos clob y blob de SQL, que pueden ser más grandes que un bloque de disco y de tamaño de varios gigabytes. Estos objetos se pueden almacenar dividiéndolos en secuencias de registros más pequeños que se organizan en un archivo B<sup>+</sup>. Los registros se pueden numerar secuencialmente o por

el desplazamiento en bytes del registro dentro del objeto, y este número se puede usar como clave de búsqueda.

### 12.3.5 Índices sobre cadenas de caracteres

La creación de índices de árboles B<sup>+</sup> sobre atributos de tipo cadena de caracteres plantea dos problemas. El primero es que las cadenas pueden ser de longitud variable. El segundo es que pueden ser largas, lo que produce un grado de salida bajo y a una altura del árbol incrementada de manera acorde.

Con las claves de búsqueda de longitud variable varios nodos pueden tener diferentes grados de salida incluso estando llenos. Un nodo se debe dividir si está lleno, es decir, si no hay espacio para añadir una nueva entrada, independientemente de cuántas entradas contenga. Análogamente, los nodos se pueden fusionar o las entradas se pueden redistribuir dependiendo de la fracción de espacio que se use en los nodos, en lugar de basarse en el número máximo de entradas que el nodo pueda contener.

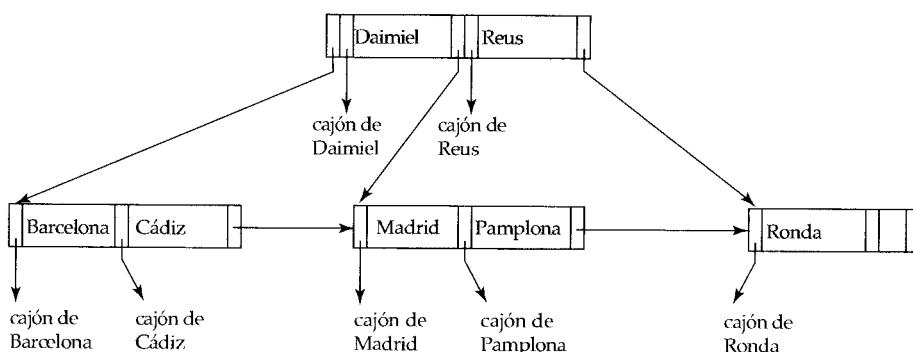
Se puede incrementar el grado de salida usando la técnica denominada **compresión del prefijo**. Con esta compresión no se almacena la clave de búsqueda completa en los nodos internos. Sólo se almacena el prefijo de la clave de búsqueda que sea suficiente para distinguir las claves de los subárboles bajo ella. Por ejemplo, si se indexa el nombre, el valor de la clave en un nodo interno podría ser un prefijo del nombre; sería suficiente almacenar “Silb” en un nodo interno en lugar del nombre completo “Silberschatz” si los valores más próximos en los dos subárboles bajo esta clave sean, por ejemplo, “Silas” y “Silver” respectivamente.

## 12.4 Archivos de índices de árbol B

Los *índices de árbol B* son similares a los índices de árbol B<sup>+</sup>. La diferencia principal entre los dos enfoques es que un árbol B elimina el almacenamiento redundante de los valores de la clave búsqueda. En el árbol B<sup>+</sup> de la Figura 12.12, las claves de búsqueda “Daimiel”, “Madrid”, “Reus” y “Pamplona” aparecen dos veces. Cada valor de clave de búsqueda aparece en algún nodo hoja; algunos se repiten en nodos internos.

Los árboles B permiten que los valores de la clave de búsqueda aparezcan solamente una vez. En la Figura 12.19 se muestra un árbol B que representa las mismas claves de búsqueda que el árbol B<sup>+</sup> de la Figura 12.12. Ya que las claves de búsqueda no están repetidas en el árbol B, sería posible almacenar el índice usando menos nodos del árbol que con el correspondiente índice de árbol B<sup>+</sup>. Sin embargo, puesto que las claves de búsqueda que aparecen en los nodos internos no aparecen en ninguna otra parte del árbol B, es necesario incluir un campo adicional para un puntero por cada clave de búsqueda de un nodo interno. Estos punteros adicionales apuntan a registros del archivo o a los cajones de la clave de búsqueda asociada.

En la Figura 12.20a aparece un nodo hoja generalizado de un árbol B; en la Figura 12.20b aparece un nodo interno. Los nodos hoja son como en los árboles B<sup>+</sup>. En los nodos internos los punteros  $P_i$  son los punteros del árbol que se utilizan también para los árboles B<sup>+</sup>, mientras que los punteros  $B_i$  en los nodos internos son punteros a cajones o registros del archivo. En la figura del árbol B generalizado hay



**Figura 12.19** Árbol B equivalente al árbol B<sup>+</sup> de la Figura 12.12.

|       |       |       |         |           |           |       |
|-------|-------|-------|---------|-----------|-----------|-------|
| $P_1$ | $K_1$ | $P_2$ | $\dots$ | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |
|-------|-------|-------|---------|-----------|-----------|-------|

(a)

|       |       |       |       |       |       |         |           |           |           |       |
|-------|-------|-------|-------|-------|-------|---------|-----------|-----------|-----------|-------|
| $P_1$ | $B_1$ | $K_1$ | $P_2$ | $B_2$ | $K_2$ | $\dots$ | $P_{m-1}$ | $B_{m-1}$ | $K_{m-1}$ | $P_m$ |
|-------|-------|-------|-------|-------|-------|---------|-----------|-----------|-----------|-------|

(b)

**Figura 12.20** Nodos típicos de un árbol B. (a) Nodo hoja. (b) Nodo interno.

$n - 1$  claves en el nodo hoja, mientras que hay  $m - 1$  claves en el nodo interno. Esta discrepancia ocurre porque los nodos internos deben incluir los punteros  $B_i$ , y de esta manera se reduce el número de claves de búsqueda que pueden contener estos nodos. Claramente,  $m < n$ , pero la relación exacta entre  $m$  y  $n$  depende del tamaño relativo de las claves de búsqueda y de los punteros.

El número de nodos a los que se accede en una búsqueda en un árbol B depende de dónde esté situada la clave de búsqueda. Una búsqueda en un árbol  $B^+$  requiere atravesar un camino desde la raíz del árbol hasta algún nodo hoja. En cambio, algunas veces es posible encontrar en un árbol B el valor deseado antes de alcanzar el nodo hoja. Sin embargo, hay que realizar aproximadamente  $n$  accesos según cuántas claves haya almacenadas tanto en el nivel de hoja de un árbol B como en los niveles internos de hoja y, dado que  $n$  es normalmente grande, la probabilidad de encontrar ciertos valores pronto es relativamente pequeña. Por otra parte, el hecho de que aparezcan menos claves de búsqueda en los nodos internos del árbol B, comparado con los árboles  $B^+$ , implica que un árbol B tiene un grado de salida menor y, por tanto, puede que tenga una profundidad mayor que la correspondiente al árbol  $B^+$ . Así, la búsqueda en un árbol B es más rápida para algunas claves de búsqueda pero más lenta para otras, aunque en general, el tiempo de la búsqueda es todavía proporcional al logaritmo del número de claves de búsqueda.

El borrado en un árbol B es más complicado. En un árbol  $B^+$  la entrada borrada siempre aparece en una hoja. En un árbol B, la entrada borrada podría aparecer en un nodo interno. El valor apropiado a colocar en su lugar se debe elegir del subárbol del nodo que contiene la entrada borrada. Concretamente, si se borra la clave de búsqueda  $K_i$ , la clave de búsqueda más pequeña que aparezca en el subárbol del puntero  $P_{i+1}$  se debe trasladar al campo ocupado anteriormente por  $K_i$ . Será necesario tomar otras medidas si ahora el nodo hoja tuviera pocas entradas. Por el contrario, la inserción en un árbol B es sólo un poco más complicada que la inserción en un árbol  $B^+$ .

Las ventajas de espacio que tienen los árboles B son escasas para índices grandes y normalmente no son de mayor importancia los inconvenientes advertidos. De esta manera, muchos implementadores de sistemas de bases de datos aprovechan la sencillez estructural de un árbol  $B^+$ . Los detalles de los algoritmos de inserción y borrado para árboles B se estudian en los ejercicios.

## 12.5 Accesos bajo varias claves

Hasta ahora se ha asumido implícitamente que se utiliza solamente un índice (o tabla asociativa) para procesar una consulta en una relación. Sin embargo, para ciertos tipos de consultas es ventajoso el uso de varios índices si existen o usar un índice construido sobre una clave de búsqueda de varios atributos.

### 12.5.1 Uso de varios índices de clave única

Considérese que el archivo *cuenta* tiene dos índices: uno para el *nombre\_sucursal* y otro para *saldo*. Dada la consulta: “Encontrar el número de todas las cuentas de la sucursal de Pamplona con saldo igual a 1.000 €”, se escribe

```
select número_préstamo
from cuenta
where nombre_sucursal = "Pamplona" and saldo = 1000
```

Hay tres estrategias posibles para procesar esta consulta:

1. Usar el índice de *nombre\_sucursal* para encontrar todos los registros correspondientes a la sucursal de Pamplona. Examinar luego esos registros para ver si *saldo* = 1.000.
2. Usar el índice de *saldo* para encontrar todos los registros pertenecientes a cuentas con un saldo de 1.000 €. Examinar luego esos registros para ver si *nombre\_sucursal* = "Pamplona".
3. Usar el índice de *nombre\_sucursal* para encontrar punteros a registros correspondientes a la sucursal de Pamplona. Usar también el índice de *saldo* para encontrar los punteros a todos los registros correspondientes a cuentas con un saldo de 1.000 €. Realizar la intersección de esos dos conjuntos de punteros. Los punteros que están en la intersección apuntan a los registros correspondientes a la vez a Pamplona y a las cuentas con un saldo de 1.000 €.

La tercera estrategia es la única de las tres que aprovecha la ventaja de tener varios índices. Sin embargo, incluso esta estrategia podría ser una mala elección si hubiese:

- Muchos registros correspondientes a la sucursal Pamplona.
- Muchos registros correspondientes a cuentas con un saldo de 1.000 €.
- Sólo unos cuantos registros pertenecientes tanto a la sucursal de Pamplona como a las cuentas con un saldo de 1.000 €.

Si ocurriera estas condiciones se tendrían que examinar un gran número de punteros para producir un resultado pequeño. La estructura de índices denominada "índice de mapas de bits" acelera significativamente la operación de inserción usada en la tercera estrategia. Los índices de mapas de bits se describen en el Apartado 12.9.

## 12.5.2 Índices sobre varias claves

Una estrategia más eficiente para este caso es crear y utilizar un índice con una clave de búsqueda (*nombre\_sucursal, saldo*)—esto es, la clave de búsqueda consistente en el nombre de la sucursal concatenado con el saldo de la cuenta. Esta clave de búsqueda, que contiene más de un atributo, se denomina a veces **clave de búsqueda compuesta**. La estructura del índice es la misma que para cualquier otro índice, con la única diferencia de que la clave de búsqueda no es un simple atributo sino una lista de atributos. La clave de búsqueda se puede representar como una tupla de valores, de la forma  $(a_1, \dots, a_n)$ , donde los atributos indexados son  $A_1, \dots, A_n$ . El orden de los valores de la clave de búsqueda es el *orden lexicográfico*. Por ejemplo, para el caso de dos atributos en la clave de búsqueda,  $(a_1, a_2) < (b_1, b_2)$  si  $a_1 < b_1$  o bien  $a_1 = b_1$  y  $a_2 < b_2$ . El orden lexicográfico es básicamente el mismo que el alfabetico.

Se puede usar un índice ordenado (árbol B<sup>+</sup>) para responder eficientemente consultas de la forma:

```
select número_préstamo
from cuenta
where nombre_sucursal = 'Pamplona' and saldo = 1000
```

Las consultas como la siguiente, que especifica una condición de igualdad sobre el primer atributo de la clave de búsqueda (*nombre\_sucursal*) y un rango sobre el segundo (*saldo*) se pueden también tratar eficientemente ya que corresponden a una consulta por rangos sobre el atributo de búsqueda.

```
select número_préstamo
from cuenta
where nombre_sucursal = 'Pamplona' and saldo < 1000
```

Incluso se puede utilizar un índice ordenado sobre la clave de búsqueda (*nombre\_sucursal, saldo*) para responder de manera eficiente a la siguiente consulta sobre un solo atributo:

```
select número_préstamo
from cuenta
where nombre_sucursal = 'Pamplona'
```

La condición de igualdad *nombre\_sucursal* = “Pamplona” es equivalente a una consulta por rangos sobre el rango con extremo inferior (Pamplona,  $-\infty$ ) y superior (Pamplona,  $+\infty$ ). Las consultas por rangos sobre sólo el atributo *nombre\_sucursal* se pueden tratar de forma similar.

Sin embargo, el uso de una estructura de índice ordenado con múltiples atributos presenta algunas deficiencias. Como ilustración considérese la consulta:

```
select número_préstamo
from cuenta
where nombre_sucursal < “Pamplona” and saldo = 1000
```

Se puede responder a esta consulta usando un índice ordenado con la clave de búsqueda (*nombre\_sucursal*, *saldo*) de la manera siguiente: para cada valor de *nombre\_sucursal* que está alfabéticamente por delante de “Pamplona”, hay que localizar los registros con un *saldo* de 1.000. Sin embargo, debido a la ordenación de los registros en el archivo, es probable que cada registro esté en un bloque de disco diferente, lo que genera muchas operaciones de E/S.

La diferencia entre esta consulta y la anterior es que la condición sobre *nombre\_sucursal* es de comparación y no de igualdad. La condición no se corresponde con una consulta por rangos sobre la clave de búsqueda.

Para acelerar el procesamiento en general de consultas con claves de búsqueda compuestas (que pueden implicar una o más operaciones de comparación) se pueden emplear varias estructuras especiales. En el Apartado 12.9.3 se considerará la estructura de *índices de mapas de bits*. Existe otra estructura, denominada *árbol R*, que también se puede usar para este propósito. Los árboles R son una extensión de los árboles B<sup>+</sup> para el tratamiento de índices en varias dimensiones. Dado que se emplean fundamentalmente con datos de tipo geográfico, su estructura se describe en el Capítulo 23.

### 12.5.3 Claves de búsqueda duplicadas

La creación de cajones de punteros para el tratamiento de las claves de búsqueda duplicadas (es decir, claves sobre archivos que puedan tener más de un registro con el mismo valor de la clave) plantea varias complicaciones cuando se implementan árboles B<sup>+</sup>. Si los cajones se guardan en un nodo hoja se necesita código extra para tratar cajones de tamaño variable y para tratar el caso en que crecen por encima del tamaño del nodo hoja. Si los cajones se almacenan en páginas separadas puede ser necesario una operación E/S adicional para acceder a los registros.

Una solución sencilla a este problema, empleada por la mayoría de los sistemas de bases de datos, es hacer que las claves de búsqueda sean únicas añadiendo un atributo único adicional a la clave. El valor de este atributo podría ser un identificador de registro (si el sistema de bases de datos les da soporte) o simplemente un número único para todos los registros con el mismo valor de la clave de búsqueda. Por ejemplo, si se tuviese un índice sobre el atributo *nombre\_cliente* de la tabla *impositor*, las entradas correspondientes a un cliente en particular tendrían diferentes valores para el atributo adicional. Así se garantizaría que la clave de búsqueda extendida fuese única.

Las búsquedas con el atributo original de la clave de búsqueda se convierten en búsquedas por rangos en la clave de búsqueda extendida, como se vio en el Apartado 12.5.2; al realizar la búsqueda, se ignora el valor del atributo adicional.

### 12.5.4 Índices de cobertura

Los **índices de cobertura** almacenan los valores de algunos atributos (distintos de los atributos de la clave de búsqueda) junto con los punteros a los registros. El almacenamiento de valores de atributo adicionales es útil en los índices secundarios, ya que permite responder algunas consultas utilizando únicamente el índice, sin ni siquiera buscar en los registros de datos.

Por ejemplo, supóngase un índice sin agrupación sobre el atributo *número\_cuenta* de la relación *cuenta*. Si se almacena el valor del atributo *saldo* junto con el puntero a registro, se pueden responder consultas que soliciten el saldo (sin el otro atributo, *nombre\_sucursal*) sin acceder al registro de *cuenta*.

Se obtendría el mismo efecto creando un índice sobre la clave de búsqueda (*número\_cuenta*, *saldo*), pero los índices de cobertura reducen el tamaño de las claves de búsqueda, lo que permite un mayor grado de salida en los nodos internos y puede reducir la altura del índice.

## 12.5.5 Índices secundarios y reubicación de registros

Algunas organizaciones de archivos, como los árboles B<sup>+</sup>, pueden cambiar la ubicación de los registros aunque éstos no se hayan modificado. A modo de ejemplo, considérese que cuando en una organización de archivo de árbol B<sup>+</sup> se divide una página hoja, varios registros se trasladan a una nueva página. En esos casos hay que actualizar todos los índices secundarios que almacenan punteros a los registros reubicados, aunque sus contenidos no hayan cambiado. Cada página hoja puede contener gran número de registros, y cada uno de ellos puede estar en diferentes ubicaciones de cada índice secundario. Así, la división de una página hoja puede exigir decenas o incluso centenas de operaciones de E/S para actualizar todos los índices secundarios afectados, lo que puede convertirla en una operación muy costosa.

A continuación se explica una técnica para resolver este problema. En los índices secundarios, en lugar de punteros a los registros indexados, se almacenan los valores de los atributos de la clave de búsqueda del índice primario. Por ejemplo, supóngase que se tiene un índice primario sobre el atributo *número\_cuenta* de la relación *cuenta*; un índice secundario sobre *nombre\_sucursal* almacenaría con cada nombre de sucursal una lista de valores de *número\_cuenta* de los registros correspondientes, en lugar de almacenar punteros a los registros.

Por tanto, la reubicación de los registros debido a la división de las páginas hoja no exige ninguna actualización de los índices secundarios. Sin embargo, la ubicación de un registro mediante el índice secundario exige ahora dos pasos: primero se utiliza el índice secundario para buscar los valores de la clave de búsqueda del índice primario y luego se utiliza el índice primario para buscar los registros correspondientes.

Este enfoque reduce en gran medida el coste de actualización de los índices debido a la reorganización de los archivos, aunque incrementa el coste de acceso a los datos mediante un índice secundario.

## 12.6 Asociación estática

Un inconveniente de la organización de archivos secuenciales es que hay que acceder a una estructura de índices para localizar los datos o utilizar una búsqueda binaria y, como resultado, más operaciones de E/S. La organización de archivos basada en la técnica de **asociación** (hashing) permite evitar el acceso a la estructura de índice. La asociación también proporciona una forma de construir índices. En los apartados siguientes se estudian las organizaciones de archivos y los índices basados en asociación.

En esta descripción de la asociación, se usará el término **cajón** (bucket) para indicar una unidad de almacenamiento que puede guardar uno o más registros. Un cajón es normalmente un bloque de disco, aunque también se podría elegir de tamaño mayor o menor que un bloque de disco.

Formalmente, sea  $K$  el conjunto de todos los valores de clave de búsqueda y sea  $B$  el conjunto de todas las direcciones de cajón. Una **función de asociación**  $h$  es una función de  $K$  a  $B$ . Sea  $h$  una función asociación.

Para insertar un registro con clave de búsqueda  $K_i$ , se calcula  $h(K_i)$ , lo que proporciona la dirección del cajón para ese registro. De momento se supone que hay espacio en el cajón para almacenar el registro. Luego, el registro se almacena en ese cajón.

Para realizar una búsqueda con el valor  $K_i$  de la clave de búsqueda, basta con calcular  $h(K_i)$  y buscar luego el cajón con esa dirección. Supóngase que dos claves de búsqueda,  $K_5$  y  $K_7$ , tienen el mismo valor de asociación; es decir,  $h(K_5) = h(K_7)$ . Si se realiza una búsqueda en  $K_5$ , el cajón  $h(K_5)$  contendrá registros con valores de la clave de búsqueda  $K_5$  y registros con valores de la clave de búsqueda  $K_7$ . Por tanto, hay que comprobar el valor de clave de búsqueda de todos los registros del cajón para asegurarse de que el registro es el deseado.

El borrado es igual de sencillo. Si el valor de la clave de búsqueda del registro que se debe borrar es  $K_i$ , se calcula  $h(K_i)$ , después se busca el cajón correspondiente a ese registro y se borra el registro del cajón.

La asociación se puede usar para dos propósitos diferentes. En la **organización de archivo asociativo** se obtiene directamente la dirección del bloque de disco que contiene el registro deseado calculando una función del valor de la clave de búsqueda del registro. En la **organización de índice asociativo** se organizan las claves de búsqueda con sus punteros asociados en una estructura de archivo asociativo.

### 12.6.1 Funciones de asociación

La peor función de asociación posible asigna todos los valores de la clave de búsqueda al mismo cajón. Una función así no es deseable, ya que hay que guardar todos los registros en el mismo cajón. Durante una búsqueda hay que examinar todos esos registros hasta encontrar el deseado. Una función de asociación ideal distribuye las claves almacenadas uniformemente entre todos los cajones, de modo que cada uno de ellos tenga el mismo número de registros.

Puesto que durante la etapa de diseño no se conocen con precisión los valores de la clave de búsqueda que se almacenarán en el archivo, se pretende elegir una función de asociación que asigne los valores de la clave de búsqueda a los cajones de manera que la distribución tenga las propiedades siguientes:

- Distribución *uniforme*. Esto es, la función de asociación asigna a cada cajón el mismo número de valores de la clave de búsqueda dentro del conjunto de *todos* los valores posibles de la misma.
- Distribución *aleatoria*. Esto es, en el caso promedio, cada cajón tendrá asignado casi el mismo número de valores, independientemente de la distribución real de los valores de la clave de búsqueda. Para ser más exactos, el valor de asociación no estará correlacionado con ningún ordenamiento de los valores de la clave de búsqueda visible exteriormente como, por ejemplo, el orden alfabético o el orden determinado por la longitud de las claves de búsqueda; parecerá que la función de asociación es aleatoria.

Como ilustración de estos principios, se escogerá una función de asociación para el archivo *cuenta* que utilice la clave búsqueda *nombre\_sucursal*. La función de asociación que se escoja debe tener las propiedades deseadas no sólo para el ejemplo del archivo *cuenta* que se ha estado utilizando, sino también para el archivo *cuenta* de tamaño real de un gran banco con muchas sucursales.

Supóngase que se decide tener 26 cajones y se define una función de asociación que asigna a los nombres que empiezan con la letra *i*-ésima del alfabeto el cajón *i*-ésimo. Esta función de asociación tiene la virtud de la simplicidad, pero no logra proporcionar una distribución uniforme, ya que se espera que haya más nombres de sucursales que comiencen con letras como B y R que con Q y con X, por ejemplo.

Supóngase ahora que se desea una función de asociación en la clave de búsqueda *saldo*. Supóngase que el saldo mínimo es 1, el saldo máximo es 100.000 y se utiliza una función de asociación que divide el valor en 10 rangos, 1—10.000, 10.001—20.000, y así sucesivamente. La distribución de los valores de la clave de búsqueda es uniforme (ya que cada cajón tiene el mismo número de valores del *saldo* diferentes), pero no es aleatoria. Los registros con saldos entre 1 y 10.000 son más frecuentes que los registros con saldos entre 90.001 y 100.000. En consecuencia, la distribución de los registros no es uniforme—algunos cajones reciben más registros que otros. Si la función presenta una distribución aleatoria, aunque se dieran esas correlaciones entre las claves de búsqueda, la aleatoriedad de la distribución haría, muy probablemente, que todos los cajones tuvieran más o menos el mismo número de registros, siempre y cuando cada clave de búsqueda apareciera sólo en una pequeña parte de los registros (si una sola clave de búsqueda aparece en gran parte de registros, es probable que el cajón que la contiene contenga más registros que otros cajones, independientemente de la función de asociación empleada).

Las funciones de asociación habituales realizan cálculos sobre la representación binaria interna de la máquina de los caracteres de la clave de búsqueda. Una función de asociación sencilla de este tipo calcula en primer lugar la suma de las representaciones binarias de los caracteres de la clave y, luego, devuelve el resto de la división entre la suma y el número de cajones. En la Figura 12.21 se muestra la aplicación de este esquema, con 10 cajones, al archivo *cuenta*, con la suposición de que la letra *i*-ésima del alfabeto está representada por el número entero *i*.

La siguiente función de asociación se puede usar para asociar una cadena en una implementación Java.

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n - 1]$$

La función se puede implementar eficientemente estableciendo el valor asociativo inicialmente como 0 e iterando desde el primero hasta el último carácter de la cadena, multiplicando en cada paso el valor asociativo por 31 y añadiendo el siguiente carácter (tratado como un entero). El resto de la división entre el resultado de esta función y el número de cajones se puede usar para la indexación.

|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|-----------|-----|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|----------|----------------------------------------------------------------|-------|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------|----------|-----|-------|---------|-----|--|--|--|
| cajón 0 | <table border="1"><tr><td></td><td></td><td></td></tr></table>                                                                                                       |       |           |     | cajón 5 | <table border="1"><tr><td>C-102</td><td>Pamplona</td><td>400</td></tr><tr><td>C-201</td><td>Pamplona</td><td>900</td></tr><tr><td>C-218</td><td>Pamplona</td><td>700</td></tr><tr><td></td><td></td><td></td></tr></table> | C-102 | Pamplona | 400                                                            | C-201 | Pamplona | 900                                                                                                                                                                  | C-218 | Pamplona | 700 |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-102   | Pamplona                                                                                                                                                             | 400   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-201   | Pamplona                                                                                                                                                             | 900   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-218   | Pamplona                                                                                                                                                             | 700   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| cajón 1 | <table border="1"><tr><td></td><td></td><td></td></tr></table>                                                                                                       |       |           |     | cajón 6 | <table border="1"><tr><td></td><td></td><td></td></tr></table>                                                                                                                                                             |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| cajón 2 | <table border="1"><tr><td></td><td></td><td></td></tr></table>                                                                                                       |       |           |     | cajón 7 | <table border="1"><tr><td>C-215</td><td>Mianus</td><td>700</td></tr><tr><td></td><td></td><td></td></tr></table>                                                                                                           | C-215 | Mianus   | 700                                                            |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-215   | Mianus                                                                                                                                                               | 700   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| cajón 3 | <table border="1"><tr><td>C-217</td><td>Barcelona</td><td>750</td></tr><tr><td>C-305</td><td>Ronda</td><td>350</td></tr><tr><td></td><td></td><td></td></tr></table> | C-217 | Barcelona | 750 | C-305   | Ronda                                                                                                                                                                                                                      | 350   |          |                                                                |       | cajón 8  | <table border="1"><tr><td>C-101</td><td>Daimiel</td><td>500</td></tr><tr><td>C-110</td><td>Daimiel</td><td>600</td></tr><tr><td></td><td></td><td></td></tr></table> | C-101 | Daimiel  | 500 | C-110 | Daimiel | 600 |  |  |  |
| C-217   | Barcelona                                                                                                                                                            | 750   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-305   | Ronda                                                                                                                                                                | 350   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-101   | Daimiel                                                                                                                                                              | 500   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-110   | Daimiel                                                                                                                                                              | 600   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| cajón 4 | <table border="1"><tr><td>C-222</td><td>Reus</td><td>700</td></tr><tr><td></td><td></td><td></td></tr></table>                                                       | C-222 | Reus      | 700 |         |                                                                                                                                                                                                                            |       | cajón 9  | <table border="1"><tr><td></td><td></td><td></td></tr></table> |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
| C-222   | Reus                                                                                                                                                                 | 700   |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |
|         |                                                                                                                                                                      |       |           |     |         |                                                                                                                                                                                                                            |       |          |                                                                |       |          |                                                                                                                                                                      |       |          |     |       |         |     |  |  |  |

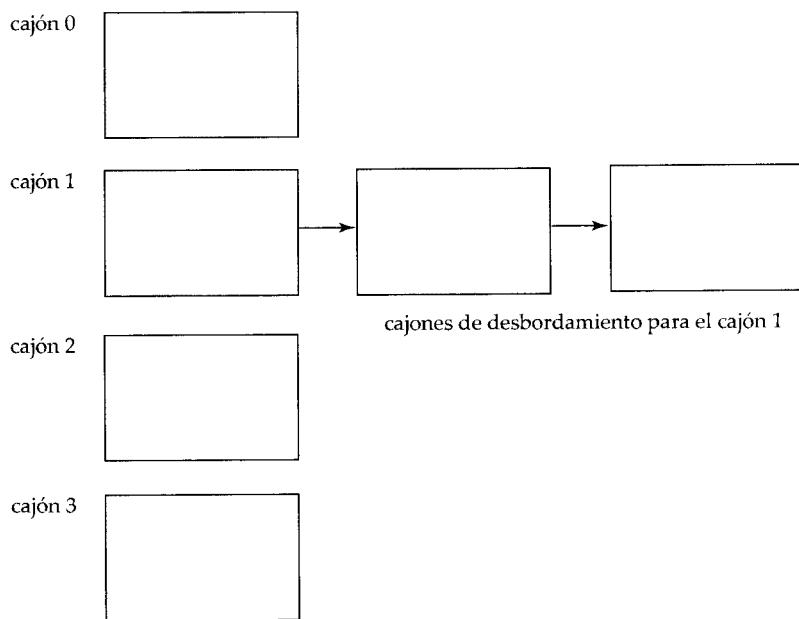
**Figura 12.21** Organización asociativa del archivo *cuenta* utilizando *nombre\_sucursal* como clave.

Las funciones de asociación requieren un diseño cuidadoso. Una mala función de asociación puede hacer que la búsqueda tarde un tiempo proporcional al número de claves de búsqueda del archivo. Una función bien diseñada da un tiempo de búsqueda para casos promedio que es una constante (pequeña), independiente del número de claves de búsqueda del archivo.

## 12.6.2 Gestión de desbordamientos de cajones

Hasta ahora se ha asumido que cuando se inserta un registro, el cajón al que se asigna tiene espacio para almacenarlo. Si el cajón no tiene suficiente espacio, sucede lo que se denomina **desbordamiento de cajones**. Los desbordamientos de cajones se pueden producir por varias razones:

- **Cajones insuficientes.** Se debe elegir un número de cajones, que se denota con  $n_B$ , tal que  $n_B > n_r/f_r$ , donde  $n_r$  denota el número total de registros que se van a almacenar y  $f_r$  denota el número de registros que caben en cada cajón. Esta designación, por supuesto, supone que se conoce el número total de registros en el momento de definir la función de asociación.
- **Atasco.** Algunos cajones tienen asignados más registros que otros, por lo que algún cajón se puede desbordar, aunque otros cajones tengan todavía espacio libre. Esta situación se denomina **atasto de cajones**. El atasco puede producirse por dos motivos:
  1. Puede que varios registros tengan la misma clave de búsqueda.
  2. Puede que la función de asociación elegida genere una distribución no uniforme de las claves de búsqueda.



**Figura 12.22** Cadena de desbordamiento en una estructura asociativa.

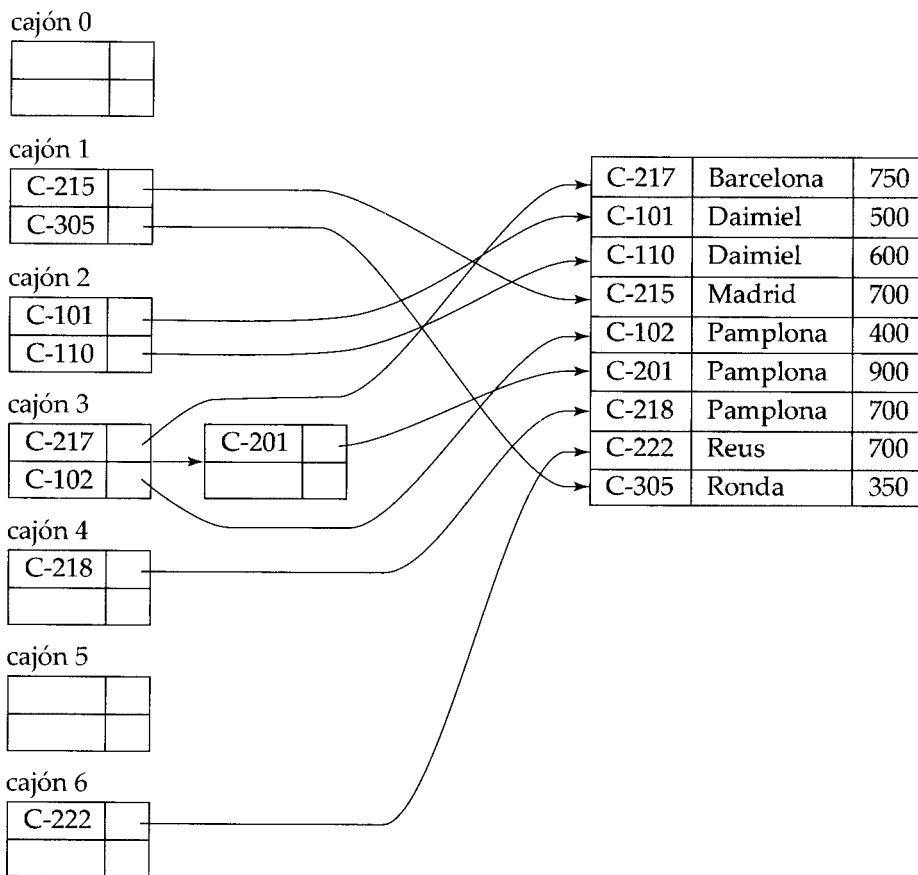
Para que la probabilidad de desbordamiento de cajones se reduzca, se escoge un número de cajones igual a  $(n_r/f_r) * (1 + d)$ , donde  $d$  es un factor de corrección, normalmente cercano a 0.2. Se pierde algo de espacio: alrededor del veinte por ciento del espacio en los cajones queda vacío. Pero la ventaja es que la probabilidad de desbordamiento se reduce.

Pese a la asignación de unos pocos cajones más de los necesarios, todavía se puede producir el desbordamiento de los cajones. El desbordamiento de los cajones se trata mediante **cajones de desbordamiento**. Si hay que insertar un registro en un cajón  $c$  y  $c$  está ya lleno, el sistema proporcionará un cajón de desbordamiento para  $c$  y el registro se insertará en ese cajón de desbordamiento. Si el cajón de desbordamiento también se encuentra lleno, el sistema proporcionará otro cajón de desbordamiento, y así sucesivamente. Todos los cajones de desbordamiento de un cajón determinado se encadenan en una lista enlazada, como se muestra en la Figura 12.22. El tratamiento del desbordamiento mediante una lista enlazada, se denomina **cadena de desbordamiento**.

Para tratar la cadena de desbordamiento es necesario modificar ligeramente el algoritmo de búsqueda. Como se dijo antes, el sistema utiliza la función de asociación sobre la clave de búsqueda para identificar un cajón  $c$ . Luego debe examinar todos los registros del cajón  $c$  para ver si coinciden con la clave de búsqueda, como antes. Además, si el cajón  $c$  tiene cajones de desbordamiento, también hay que examinar los registros de todos los cajones de desbordamiento.

La forma de la estructura asociativa que se acaba de describir se denomina a veces **asociación cerrada**. En una aproximación alternativa, conocida como **asociación abierta**, se fija el conjunto de cajones y no hay cadenas de desbordamiento. En su lugar, si un cajón está lleno, el sistema inserta los registros en algún otro cajón del conjunto inicial  $C$  de cajones. Un criterio es utilizar el siguiente cajón (en orden cíclico) que tenga espacio; esta política se llama *ensayo lineal*. Se utilizan también otros criterios, como el cálculo de funciones de asociación adicionales. La asociación abierta se emplea en la construcción de tablas de símbolos para compiladores y ensambladores, pero la asociación cerrada es preferible para los sistemas de bases de datos. El motivo es que el borrado con asociación abierta es problemático. Normalmente, los compiladores y los ensambladores sólo realizan operaciones de búsqueda e inserción en sus tablas de símbolos. Sin embargo, en los sistemas de bases de datos es importante poder tratar tanto el borrado como la inserción. Por tanto, la asociatividad abierta sólo tiene una importancia menor en la implementación de bases de datos.

Un inconveniente importante de la forma de asociación que se ha descrito, es que hay que elegir la función de asociación cuando se implementa el sistema y no se puede cambiar fácilmente después si el archivo que se está indexando aumenta o disminuye de tamaño. Como la función  $h$  asigna valores



**Figura 12.23** Índice asociativo de la clave de búsqueda *número\_cuenta* del archivo *cuenta*.

de la clave búsqueda a un conjunto fijo  $C$  de direcciones de cajón, se desperdicia espacio si se hace  $C$  de gran tamaño para manejar el futuro crecimiento del archivo. Si  $C$  es demasiado pequeño, cada cajón contendrá registros de muchos valores de la clave de búsqueda y se pueden provocar desbordamientos de cajones. A medida que el archivo aumenta de tamaño, el rendimiento se degrada. Más adelante, en el Apartado 12.7, se estudiará la manera de cambiar dinámicamente el número de cajones y la función de asociación.

### 12.6.3 Índices asociativos

La asociatividad se puede utilizar no solamente para la organización de archivos, sino también para la creación de estructuras de índice. Cada **índice asociativo** (hash index) organiza las claves de búsqueda, con sus punteros asociados, en una estructura de archivo asociativo. Los índices asociativos se construyen como se indica a continuación. Primero se aplica una función de asociación sobre la clave de búsqueda para identificar un cajón, luego se almacenan la clave y los punteros asociados en el cajón (o en los cajones de desbordamiento). En la Figura 12.23 se muestra un índice asociativo secundario del archivo *cuenta* para la clave de búsqueda *número\_cuenta*. La función de asociación utilizada calcula la suma de las cifras del número de cuenta módulo siete. El índice asociativo posee siete cajones, cada uno de tamaño dos (un índice realista tendría, por supuesto, cajones de mayor tamaño). Uno de los cajones tiene tres claves asignadas, por lo que tiene un cajón de desbordamiento. En este ejemplo, *número\_cuenta* es clave primaria de *cuenta*, por lo que cada clave de búsqueda sólo tiene asociado un puntero. En general, se pueden asociar varios punteros con cada clave.

Se usa el término **índice asociativo** para denotar las estructuras de archivo asociativo, así como los índices secundarios asociativos. Estrictamente hablando, los índices asociativos son sólo estructuras de índices secundarios. Los índices asociativos no se necesitan nunca como estructuras de índices con agru-

pación ya que, si un archivo está organizado mediante asociatividad, no hay necesidad de ninguna estructura de índice asociativo adicional. Sin embargo, como la organización en archivos asociativos proporciona el mismo acceso directo a los registros que el indexado, se finge que la organización de un archivo mediante asociación también tiene en él un índice con agrupación asociativo.

## 12.7 Asociación dinámica

Como se ha visto, la necesidad de fijar el conjunto  $C$  de direcciones de los cajones presenta un problema serio con la técnica de asociación estática vista en el apartado anterior. La mayor parte de las bases de datos aumenta de tamaño con el tiempo. Si se va a utilizar la asociación estática para esas bases de datos, hay tres opciones:

1. Elegir una función de asociación de acuerdo con el tamaño actual del archivo. Esta opción provocará una degradación del rendimiento a medida que la base de datos aumente de tamaño.
2. Elegir una función de asociación de acuerdo con el tamaño previsto del archivo en un momento determinado del futuro. Aunque se evite la degradación del rendimiento, puede que inicialmente se pierda una cantidad de espacio significativa.
3. Reorganizar periódicamente la estructura asociativa en respuesta al crecimiento del archivo. Esta reorganización supone elegir una nueva función de asociación, volver a calcular la función de asociación de cada registro del archivo y generar nuevas asignaciones de cajones. Esta reorganización es una operación masiva que consume mucho tiempo. Además, hay que prohibir el acceso al archivo durante la reorganización.

Algunas técnicas de **asociación dinámica** permiten modificar dinámicamente la función de asociación para adaptarse al aumento o disminución de tamaño de la base de datos. En esta apartado se describe una forma de asociación dinámica, denominada **asociación extensible**. Las notas bibliográficas proporcionan referencias a otras formas de asociación dinámica.

### 12.7.1 Estructura de datos

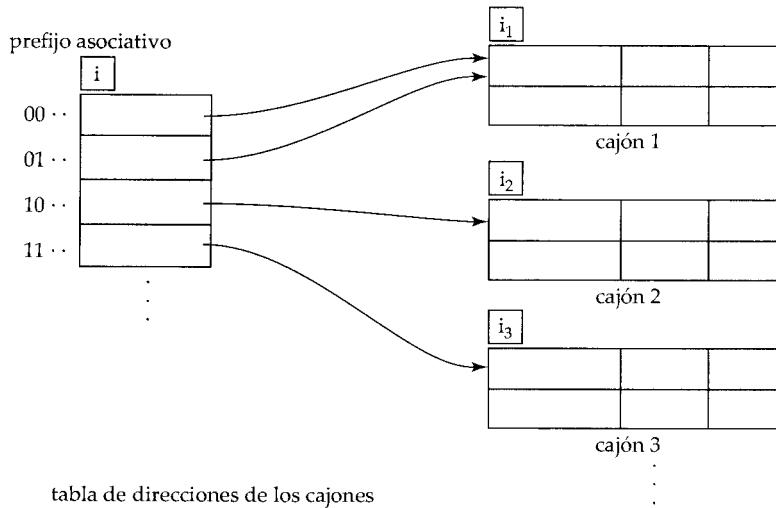
La asociación extensible hace frente a los cambios del tamaño de la base de datos dividiendo y fusionando los cajones a medida que la base de datos aumenta o disminuye. En consecuencia, se conserva la eficiencia espacial. Además, puesto que la reorganización sólo se lleva a cabo en un cajón simultáneamente, la degradación del rendimiento resultante es aceptablemente baja.

Con la asociación extensible se elige una función de asociación  $h$  con las propiedades deseadas de uniformidad y aleatoriedad. Sin embargo, esa función de asociación genera valores dentro de un rango relativamente amplio—por ejemplo, los enteros binarios de  $b$  bits. Un valor normal de  $b$  es 32.

No se crea un cajón para cada valor de la función de asociación. De hecho,  $2^{32}$  es más de cuatro mil millones, y no son razonables tantos cajones salvo para las mayores bases de datos. En vez de eso, se crean cajones bajo demanda, a medida que se insertan registros en el archivo. Inicialmente no se utilizan todos los  $b$  bits del valor de la función de asociación. En cualquier momento se utilizan  $i$  bits, donde  $0 \leq i \leq b$ . Esos  $i$  bits son utilizados como reserva en una tabla adicional de direcciones de los cajones. El valor de  $i$  aumenta o disminuye con el tamaño de la base de datos.

En la Figura 12.24 se muestra una estructura general de asociación extensible. La  $i$  que aparece en la figura encima de la tabla de direcciones de los cajones indica que se requieren  $i$  bits del valor de la función de asociación  $h(K)$  para determinar el cajón apropiado para  $K$ . Obviamente, ese número cambia a medida que el archivo aumenta de tamaño. Aunque se requieren  $i$  bits para encontrar la entrada correcta en la tabla de direcciones de los cajones, varias entradas consecutivas de la tabla pueden apuntar al mismo cajón. Todas esas entradas tendrán un prefijo de asociación común, pero la longitud de ese prefijo puede ser menor que  $i$ . Por lo tanto, se asocia con cada cajón un número entero que proporciona la longitud del prefijo de asociación común. En la Figura 12.24, el entero asociado con el cajón  $j$  aparece como  $i_j$ . El número de entradas de la tabla de direcciones de cajones que apuntan al cajón  $j$  es

$$2^{(i - i_j)}$$



**Figura 12.24** Estructura asociativa general extensible.

### 12.7.2 Consultas y actualizaciones

A continuación se estudiará la forma de realizar la búsqueda, la inserción y el borrado en una estructura asociativa extensible.

Para localizar el cajón que contiene el valor de la clave de búsqueda  $K_l$ , el sistema toma los primeros  $i$  bits más significativos de  $h(K_l)$ , busca la entrada de la tabla correspondiente a esa cadena de bits, y sigue el puntero del cajón de esa entrada de la tabla.

Para insertar un registro con un valor de la clave de búsqueda  $K_l$  se sigue el mismo procedimiento de búsqueda que antes, y se llega a un cajón—por ejemplo,  $j$ . Si hay sitio en ese cajón, se inserta en él el registro. Si, por el contrario, el cajón está lleno, hay que dividir el cajón y redistribuir los registros actuales, junto con el nuevo. Para dividir el cajón, primero hay que determinar, a partir del valor de la función de asociación, si hace falta incrementar el número de bits que hay que utilizar.

- Si  $i = i_j$ , entonces solamente apunta al cajón  $j$  una entrada de la tabla de direcciones de los cajones. Por tanto, es necesario incrementar el tamaño de la tabla de direcciones de los cajones para incluir los punteros a los dos cajones que resultan de la división del cajón  $j$ . Esto se consigue considerando otro bit más del valor de asociación. Se incrementa en uno el valor de  $i$ , lo que duplica el tamaño de la tabla de direcciones de los cajones. Cada entrada se sustituye por dos entradas, ambas con el mismo puntero que la entrada original. Ahora, dos entradas de la tabla de direcciones de cajones apuntan al cajón  $j$ . Se asigna un nuevo cajón (el cajón  $z$ ) y se hace que la segunda entrada apunte al nuevo cajón. Se definen  $i_j$  e  $i_z$  como  $i$ . A continuación se vuelve a calcular la función de asociación para todos los registros del cajón  $j$  y, en función de los primeros  $i$  bits (recuérdese que se ha añadido uno a  $i$ ), se mantienen en el cajón  $j$  o se colocan en el cajón recién creado.

Ahora se vuelve a intentar la inserción del nuevo registro. Normalmente el intento tiene éxito. Sin embargo, si todos los registros del cajón  $j$ , así como el registro nuevo, tienen el mismo prefijo de asociación, será necesario volver a dividir el cajón, ya que tanto los registros del cajón  $j$  como el registro nuevo tienen asignado el mismo cajón. Si la función de asociación se eligió cuidadosamente, es poco probable que una simple inserción provoque que un cajón se divida más de una vez, a menos que haya un gran número de registros con la misma clave de búsqueda. Si todos los registros del cajón  $j$  tienen el mismo valor de la clave de búsqueda, la división no servirá de nada. En esos casos se utilizan cajones de desbordamiento para almacenar los registros, como en la asociación estática.

- Si  $i > i_j$ , entonces más de una entrada en la tabla de direcciones de los cajones apunta al cajón  $j$ . Por tanto, se puede dividir el cajón  $j$  sin aumentar el tamaño de la tabla de direcciones

|       |           |     |
|-------|-----------|-----|
| C-217 | Barcelona | 750 |
| C-101 | Daimiel   | 500 |
| C-110 | Daimiel   | 600 |
| C-215 | Madrid    | 700 |
| C-102 | Pamplona  | 400 |
| C-201 | Pamplona  | 900 |
| C-218 | Pamplona  | 700 |
| C-222 | Reus      | 700 |
| C-305 | Ronda     | 350 |

**Figura 12.25** Archivo *cuenta* de ejemplo.

| <i>nombre_sucursal</i> | $h(\text{nombre\_sucursal})$            |
|------------------------|-----------------------------------------|
| Barcelona              | 0010 1101 1111 1011 0010 1100 0011 0000 |
| Daimiel                | 1010 0011 1010 0000 1100 0110 1001 1111 |
| Madrid                 | 1100 0111 1110 1101 1011 1111 0011 1010 |
| Pamplona               | 1111 0001 0010 0100 1001 0011 0110 1101 |
| Reus                   | 0011 0101 1010 0110 1100 1001 1110 1011 |
| Ronda                  | 1101 1000 0011 1111 1001 1100 0000 0001 |

**Figura 12.26** Función de asociación para *nombre\_sucursal*.

de los cajones. Obsérvese que todas las entradas que apuntan al cajón  $j$  corresponden a prefijos de asociación que tienen el mismo valor en los  $i_j$  bits situados más a la izquierda. Se asigna un nuevo cajón (el cajón  $z$ ) y se definen  $i_j$  e  $i_z$  con el valor que resulta de añadir uno al valor original de  $i_j$ . A continuación hay que ajustar las entradas de la tabla de direcciones de los cajones que anteriormente apuntaban al cajón  $j$  (obsérvese que, con el nuevo valor de  $i_j$ , no todas las entradas corresponden a prefijos de asociación que tienen el mismo valor en los  $i_j$  bits situados más a la izquierda). La primera mitad de las entradas se deja como estaba (apuntando al cajón  $j$ ) y se hace que el resto de las entradas apunte al cajón recién creado (el cajón  $z$ ). Luego, como en el caso anterior, se vuelve a calcular la función de asociación para todos los registros del cajón  $j$  y se asignan o bien al cajón  $j$  o bien al cajón  $z$  recién creado.

Luego se vuelve a intentar la inserción. En el improbable caso de que vuelva a fallar, se aplica uno de los dos casos,  $i = i_j$  o  $i > i_j$ , según corresponda.

Obsérvese que en ambos casos solamente se necesita volver a calcular la función de asociación de los registros del cajón  $j$ .

Para borrar un registro con valor de la clave de búsqueda  $K_l$  se sigue el mismo procedimiento de búsqueda, que finaliza en un cajón—por ejemplo,  $j$ . Se borran tanto el registro del archivo como la clave de búsqueda del cajón. También se elimina el cajón si se queda vacío. Obsérvese que, en este momento, se pueden fusionar varios cajones y se puede reducir el tamaño de la tabla de direcciones de los cajones a la mitad. El procedimiento para decidir los cajones que se deben fusionar y el momento de hacerlo se deja al lector como ejercicio. Las condiciones bajo las que la tabla de direcciones de los cajones se puede reducir de tamaño también se dejan como ejercicio. A diferencia de la fusión de los cajones, el cambio de tamaño de la tabla de direcciones de los cajones es una operación bastante costosa si la tabla es de gran tamaño. Por tanto, sólo merece la pena reducir el tamaño de la tabla de direcciones de los cajones si el número de cajones se reduce considerablemente.

El ejemplo del archivo *cuenta* de la Figura 12.25 ilustra la operación de inserción. Los valores de asociación de 32 bits de *nombre\_sucursal* se muestran en la Figura 12.26. Supóngase que, inicialmente, el archivo está vacío, como se muestra en la Figura 12.27. Los registros se insertan de uno en uno. Para mostrar todas las características de la asociación extensible en una estructura pequeña, se hará la suposición no realista de que cada cajón sólo puede contener dos registros.

Se va a insertar el registro (C-217, Barcelona, 750). La tabla de direcciones de los cajones contiene un puntero al único cajón existente y el sistema inserta el registro. A continuación se inserta el registro (C-101, Daimiel, 500). Este registro también se inserta en el único cajón de la estructura.

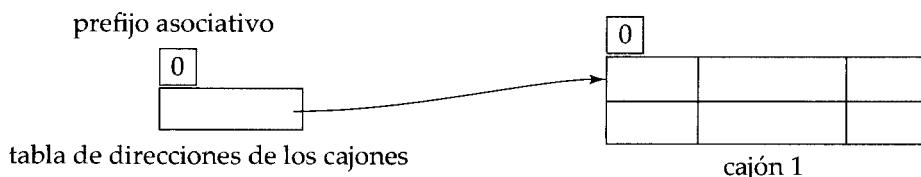
Cuando se intenta insertar el siguiente registro (C-110, Daimiel, 600), el cajón está lleno. Ya que  $i = i_0$ , es necesario incrementar el número de bits del valor de asociación que se utilizan. Ahora se utiliza un bit, lo que permite  $2^1 = 2$  cajones. Este incremento en el número de bits necesarios necesita doblar el tamaño de la tabla de direcciones de cajones a dos entradas. El sistema divide el cajón, coloca en el nuevo aquellos registros cuya clave de búsqueda tiene un valor de asociación que comienza por 1 y deja el resto de los registros en el cajón original. En la Figura 12.28 se muestra el estado de la estructura después de la división.

A continuación se inserta (C-215, Madrid, 700). Como el primer bit de  $h(\text{Madrid})$  es 1, hay que insertar ese registro en el cajón al que apunta la entrada “1” de la tabla de direcciones de los cajones. Una vez más, el cajón se encuentra lleno e  $i = i_1$ . Se incrementa a dos el número de bits del valor de asociación que se usan. Este incremento en el número de bits necesarios hace que se doble el tamaño de la tabla de direcciones de los cajones a cuatro entradas, como se muestra en la Figura 12.29. Como el cajón de la Figura 12.28 con el prefijo 0 del valor de asociación no se dividió, las dos entradas, 00 y 01, de la tabla de direcciones de los cajones apuntan a ese cajón.

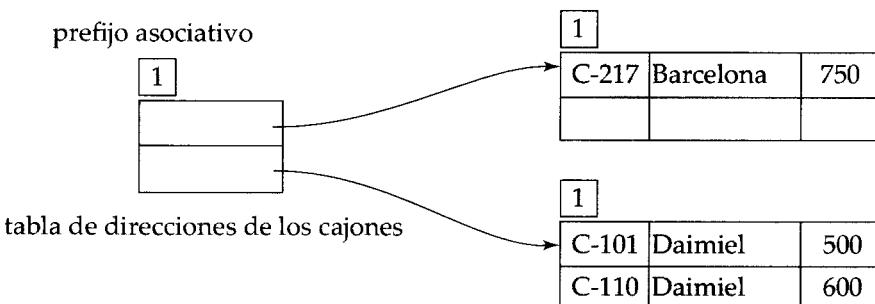
Para cada registro del cajón de la Figura 12.28 con prefijo de asociación 1 (el cajón que se va a dividir) se examinan los dos primeros bits del valor de asociación para determinar el cajón de la nueva estructura que le corresponde.

A continuación se inserta el registro (C-102, Pamplona, 400), que se aloja en el mismo cajón que Madrid. La siguiente inserción, la de (C-201, Pamplona, 900), provoca un desbordamiento en un cajón, lo que provoca el incremento del número de bits y la duplicación del tamaño de la tabla de direcciones de los cajones. La inserción del tercer registro de Pamplona, (C-218, Pamplona, 700), produce otro desbordamiento. Sin embargo, este desbordamiento no se puede resolver incrementando el número de bits, ya que hay tres registros con el mismo valor de asociación exactamente. Por tanto se utiliza un cajón de desbordamiento, como se muestra en la Figura 12.30.

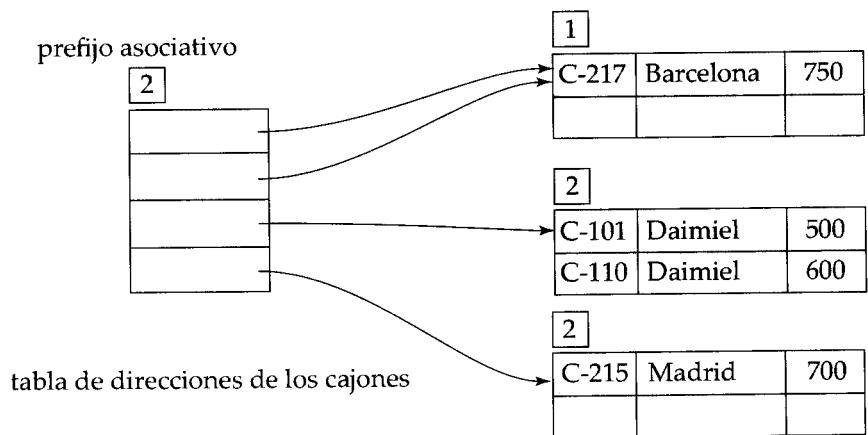
Se continúa de esta manera hasta que se hayan insertado todos los registros del archivo *cuenta* de la Figura 12.25. La estructura resultante se muestra en la Figura 12.31.



**Figura 12.27** Estructura asociativa extensible inicial.



**Figura 12.28** Estructura asociativa después de tres inserciones.

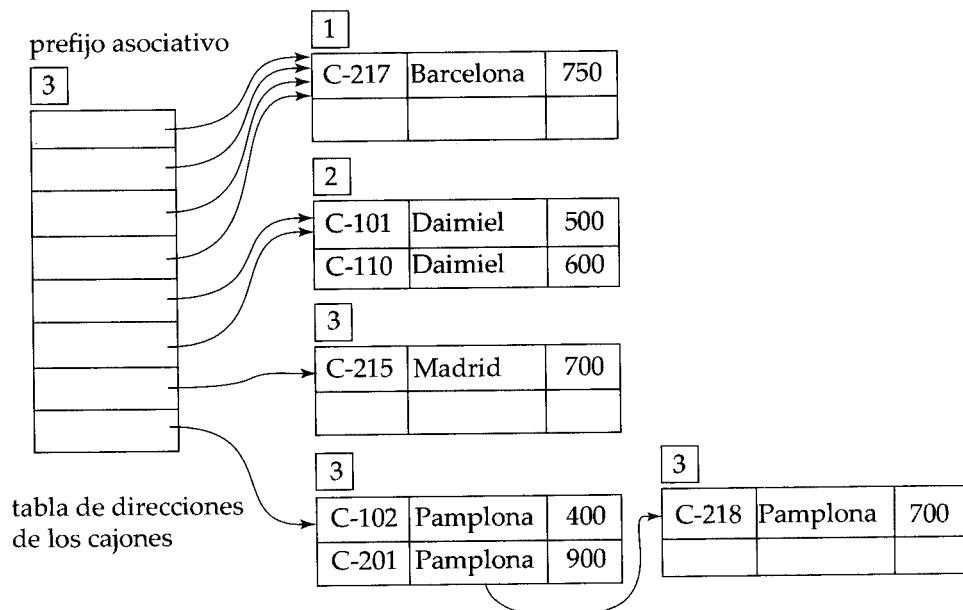


**Figura 12.29** Estructura asociativa después de cuatro inserciones.

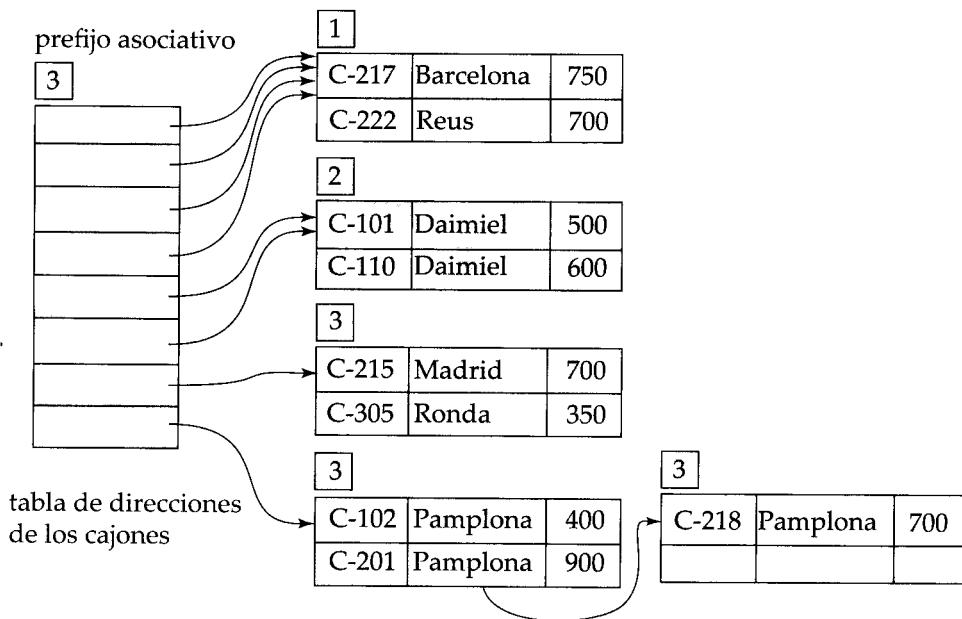
### 12.7.3 Comparación entre la asociación estática y la dinámica

A continuación se examinan las ventajas e inconvenientes de la asociación extensible respecto de la asociación dinámica. La ventaja principal de la asociación extensible es que el rendimiento no se degrada a medida que el archivo aumenta de tamaño. Además, el espacio adicional necesario es mínimo. Aunque la tabla de direcciones de los cajones provoca un gasto adicional, sólo contiene un puntero por cada valor de asociación para la longitud actual del prefijo. Por tanto, el tamaño de la tabla es pequeño. El principal ahorro de espacio de la asociación extensible respecto de otras formas de asociación es que no es necesario reservar cajones para un futuro aumento de tamaño; en vez de eso, los cajones se pueden asignar de manera dinámica.

Un inconveniente de la asociación extensible es que la búsqueda implica un nivel adicional de referencia, ya que se debe acceder a la tabla de direcciones de los cajones antes que a los propios cajones. Esta referencia adicional sólo tiene una mínima repercusión en el rendimiento. Aunque las estructuras asociativas que se estudiaron en el Apartado 12.6 no tienen este nivel adicional de referencia, pierden esa mínima ventaja de rendimiento cuando se llenan.



**Figura 12.30** Estructura asociativa después de siete inserciones.



**Figura 12.31** Estructura asociativa extensible para el archivo *cuenta*.

Por tanto, la asociación extensible se muestra como una técnica muy atractiva, siempre que se acepte la complejidad añadida de su implementación. En las notas bibliográficas se proporcionan descripciones más detalladas de la implementación de la asociación extensible.

Las notas bibliográficas también ofrecen referencias a otra forma de asociación dinámica denominada **asociación lineal**, que evita el nivel adicional de referencia asociado con la asociación extensible al posible coste de más cajones de desbordamiento.

## 12.8 Comparación de la indexación ordenada y la asociación

Se han examinado varios esquemas de indexación ordenada y varios esquemas de asociación. Se pueden organizar los archivos de registros como archivos ordenados, mediante una organización de índice secuencial u organizaciones de árbol B<sup>+</sup>. Alternativamente, se pueden organizar los archivos mediante la asociación. Finalmente, los archivos se pueden organizar como montículos, en los que los registros no están ordenados de ninguna manera en especial.

Cada esquema tiene sus ventajas, dependiendo de la situación. Un fabricante de un sistema de bases de datos puede proporcionar muchos esquemas y dejar al diseñador de la base de datos la decisión final sobre los esquemas que se utilizarán. Sin embargo, este enfoque exige que el fabricante escriba más código, lo que aumenta tanto el coste del sistema como el espacio que ocupa. La mayor parte de los sistemas de bases de datos soportan árboles B<sup>+</sup> y pueden dar soporte también a alguna forma de organización de archivos asociativos o de índices asociativos.

Para hacer una buena elección de organización de archivos y de técnica de indexado, el fabricante o el diseñador de la base de datos debe tener en consideración los siguientes aspectos:

- ¿Es aceptable el coste de una reorganización periódica del índice o de la estructura asociativa?
- ¿Cuál es la frecuencia relativa de las inserciones y de los borrados?
- ¿Es deseable optimizar el tiempo medio de acceso a expensas de incrementar el peor tiempo de acceso posible?
- ¿Qué tipos de consultas se supone que van a formular los usuarios?

De estos puntos ya se han examinado los tres primeros, comenzando con la revisión de las ventajas relativas de las distintas técnicas de indexado, y nuevamente en la discusión de las técnicas de asociación.

El cuarto punto, el tipo de consultas esperado, resulta fundamental para la elección entre la indexación ordenada y la asociación.

Si la mayoría de las consultas son de la forma:

```
select A_1, A_2, \dots, A_n
from r
where $A_i = c$
```

entonces, para procesarla, el sistema realiza una búsqueda en un índice ordenado o en una estructura asociativa del atributo  $A_i$  con el valor  $c$ . Para este tipo de consultas es preferible un esquema asociativo. Las búsquedas en índices ordenados requieren un tiempo proporcional al logaritmo del número de valores de  $A_i$  en  $r$ . Sin embargo, en las estructuras asociativas, el tiempo medio de búsqueda es una constante independiente del tamaño de la base de datos. La única ventaja de los índices respecto de las estructuras asociativas en este tipo de consultas es que el tiempo de búsqueda en el peor de los casos es proporcional al logaritmo del número de valores de  $A_i$  en  $r$ . Por el contrario, si se utiliza una estructura asociativa, el tiempo de búsqueda en el peor de los casos es proporcional al número de valores de  $A_i$  en  $r$ . Sin embargo, es poco probable en el caso de la asociación que se dé el peor caso de búsqueda posible (máximo tiempo de búsqueda) y, en este caso, es preferible emplear una estructura asociativa.

Las técnicas de índices ordenados son preferibles a las estructuras asociativas en los casos en los que la consulta especifica un rango de valores. Estas consultas tienen el siguiente aspecto:

```
select A_1, A_2, \dots, A_n
from r
where $A_i \leq c_1$ and $A_i \geq c_2$
```

En otras palabras, la consulta anterior busca todos los registros con  $A_i$  valores comprendidos entre  $c_1$  y  $c_2$ .

Considérese la manera de procesar esta consulta empleando un índice ordenado. En primer lugar se realiza una búsqueda del valor  $c_1$ . Una vez que se ha encontrado el cajón que contiene el valor  $c_1$ , se sigue el orden de la cadena de punteros del índice para leer el siguiente cajón y se continúa de esta manera hasta que se llega a  $c_2$ .

Si, en vez de un índice ordenado, se tiene una estructura asociativa, se puede llevar a cabo una búsqueda de  $c_1$  y localizar el cajón correspondiente—pero, en general, no resulta fácil determinar el cajón que hay que examinar a continuación. La dificultad surge porque una buena función de asociación asigna valores a los cajones de manera aleatoria. Por tanto, no hay un concepto sencillo de “siguiente cajón según el orden establecido”. La razón por la que no se puede encadenar un cajón detrás de otro según un cierto orden de  $A_i$  es que cada cajón tiene asignado muchos valores de la clave de búsqueda. Como los valores están diseminados aleatoriamente según la función de asociación, es probable que los valores del rango especificado estén espaciados por muchos cajones o, tal vez, por todos. Por esa razón, hay que leer todos los cajones para encontrar las claves de búsqueda necesarias.

Normalmente el diseñador elige la indexación ordenada, a menos que se sepa de antemano que las consultas de rango van a ser poco frecuentes, en cuyo caso se escoge la asociación. Las organizaciones asociativas resultan especialmente útiles para los archivos temporales creados durante el procesamiento de las consultas, siempre que se realicen búsquedas basadas en valores de la clave pero no se vayan a realizar consultas de rango.

## 12.9 Índices de mapas de bits

Los índices de mapas de bits son un tipo de índices especializado diseñado para la consulta sencilla sobre varias claves, aunque cada índice de mapas de bits se construya para una única clave.

Para que se utilicen los índices de mapas de bits, los registros de la relación deben estar numerados secuencialmente comenzando, por ejemplo, por 0. Dado un número  $n$  debe ser fácil recuperar el registro con número  $n$ . Esto resulta especialmente fácil de conseguir si los registros son de tamaño fijo y están asignados a bloques consecutivos de un archivo. El número de registro se puede traducir fácilmente en un número de bloque y en un número que identifica el registro dentro de ese bloque.

Considérese una relación  $r$  con un atributo  $A$  que sólo puede tomar como valor un número pequeño (por ejemplo, entre 2 y 20). Por ejemplo, la relación *info\_cliente* puede tener el atributo *sexo*, que sólo

puede tomar los valores m (masculino) o f (femenino). Otro ejemplo puede ser el atributo *nivel\_ingresos*, en el que los ingresos se han dividido en cinco niveles: *L1*: \$0 – 9999, *L2*: \$10.000 – 19.999, *L3*: 20.000 – 39.999, *L4*: 40.000 – 74.999 y *L5*: 75.000 –  $\infty$ . Aquí, los datos originales pueden tomar muchos valores, pero un analista de datos los ha dividido en un número pequeño de rangos para simplificar su análisis.

### 12.9.1 Estructura de los índices de mapas de bits

Un **mapa de bits** es simplemente un array de bits. En su forma más sencilla, un **índice de mapas de bits** sobre el atributo *A* de la relación *r* consiste en un mapa de bits para cada valor que pueda tomar *A*. Cada mapa de bits tiene tantos bits como el número de registros de la relación. El *i*-ésimo bit del mapa de bits para el valor *v<sub>j</sub>* se define como 1 si el registro con número *i* tiene el valor *v<sub>j</sub>* para el atributo *A*. El resto de los bits del mapa de bits se define como 0.

En este ejemplo hay un mapa de bits para el valor m y otro para f. El *i*-ésimo bit del mapa de bits para m se define como 1 si el valor *sexo* del registro con número *i* es m. El resto de bits del mapa de bits de m se definen como 0. Análogamente, el mapa de bits de f tiene el valor 1 para los bits correspondientes a los registros con el valor f para el atributo *sexo*; el resto de bits tienen el valor 0. La Figura 12.32 muestra un ejemplo de índices de mapa de bits para la relación *info\_cliente*.

Ahora se considerará cuándo resultan útiles los mapas de bits. La manera más sencilla de recuperar todos los registros con el valor m (o f) sería simplemente leer todos los registros de la relación y seleccionar los de valor m (o f, respectivamente). El índice de mapas de bits no ayuda realmente a acelerar esa selección.

De hecho, los índices de mapas de bits resultan útiles para las selecciones sobre todo cuando hay selecciones bajo varias claves. Supóngase que se crea un índice de mapas de bits sobre el atributo *nivel\_inglesos*, que ya se ha descrito antes, además del índice de mapas de bits para *sexo*.

Considérese ahora una consulta que seleccione mujeres con ingresos en el rango 10.000 – 19.999 €. Esta consulta se puede expresar como  $\sigma_{\text{sexo}=\text{f} \wedge \text{nivel\_ingresos}=\text{L2}}(r)$ . Para evaluar esta selección se busca el valor f en el mapa de bits de *sexo* y el valor L2 en el de *nivel\_inglesos* y se realiza la **intersección** (conjunction lógica) de los dos mapas de bits. En otras palabras, se calcula un nuevo mapa de bits en el que el bit *i* tenga el valor 1 si el *i*-ésimo bit de los dos mapas de bits es 1, y el valor 0 en caso contrario. En el ejemplo de la Figura 12.32, la intersección del mapa de bits de *sexo = f* (01101) y el de *nivel\_inglesos = L2* (01000) da como resultado el mapa de bits 01000.

Como el primer atributo puede tomar dos valores y el segundo cinco, se puede esperar, en promedio, que sólo de 1 a 10 registros satisfagan la condición combinada de los dos atributos. Si hay más condiciones, es probable que la proporción de los registros que satisfacen todas las condiciones sea bastante pequeña. El sistema puede calcular así el resultado de la consulta buscando todos los bits con valor 1 del mapa de bits resultado de la intersección y recuperando los registros correspondientes. Si la proporción es grande, la exploración de la relación completa seguirá siendo la alternativa menos costosa.

Otro uso importante de los mapas de bits es el recuento del número de tuplas que satisfacen una selección dada. Esas consultas son importantes para el análisis de datos. Por ejemplo, si se desea de-

| número de registro | nombre   | sexo | dirección  | nivel_inglesos |
|--------------------|----------|------|------------|----------------|
| 0                  | Juan     | m    | Perryridge | N1             |
| 1                  | Diana    | f    | Brooklyn   | N2             |
| 2                  | Maria    | f    | Jonestown  | N1             |
| 3                  | Pedro    | m    | Brooklyn   | N4             |
| 4                  | Katzalin | f    | Perryridge | N3             |

| mapas de bits para sexo | mapas de bits para nivel_inglesos |
|-------------------------|-----------------------------------|
| m      1 0 0 1 0        | N1    1 0 1 0 0                   |
| f      0 1 1 0 1        | N2    0 1 0 0 0                   |
|                         | N3    0 0 0 0 1                   |
|                         | N4    0 0 0 1 0                   |
|                         | N5    0 0 0 0 0                   |

Figura 12.32 Índices de mapas de bits para la relación *info\_cliente*.

terminar el número de mujeres que tienen un nivel de ingresos  $L_2$ , se calcula la intersección de los dos mapas de bits y luego se cuenta el número de bits de valor 1 en el mapa de bits resultado de la intersección. Así se puede obtener el resultado deseado del índice de mapa de bits sin ni siquiera acceder a la relación.

Los índices de mapas de bits son generalmente bastante pequeños en comparación con el tamaño real de la relación. Los registros suelen tener entre decenas y centenares de bytes de longitud, mientras que un solo bit representa a un registro en el mapa de bits. Por tanto, el espacio ocupado por cada mapa de bits suele ser menos del uno por ciento del espacio ocupado por la relación. Por ejemplo, si el tamaño del registro de una relación dada es de 100 bytes, el espacio ocupado por cada mapa de bits sería la octava parte del uno por ciento del espacio ocupado por la relación. Si el atributo  $A$  de la relación sólo puede tomar uno valor de entre ocho, el índice de mapas de bits de ese atributo consiste en ocho mapas de bits que, juntos, sólo ocupan el uno por ciento del tamaño de la relación.

El borrado de registros crea huecos en la secuencia de registros, ya que el desplazamiento de registros (o de los números de registro) para llenar los huecos resultaría excesivamente costoso. Para reconocer los registros borrados se puede almacenar un **mapa de bits de existencia** en el que el bit  $i$  sea 0 si el registro  $i$  no existe, y 1 en caso contrario. Se verá la necesidad de la existencia de los mapas de bits en el Apartado 12.9.2. La inserción de registros no debe afectar a la secuencia de numeración de los demás registros. Por tanto, se puede insertar tanto añadiendo registros al final del archivo como reemplazando los registros borrados.

## 12.9.2 Implementación eficiente de las operaciones de mapas de bits

Se puede calcular fácilmente la intersección de dos mapas de bits usando un bucle **for**: la iteración  $i$ -ésima del bucle calcula la conjunción (**and**) de los bits  $i$ -ésimos de los dos mapas de bits. Se puede acelerar considerablemente el cálculo de la intersección usando las instrucciones de bits **and** soportadas por la mayoría de los conjuntos de instrucciones de las computadoras. Cada *palabra* suele constar de 32 o 64 bits, en función de la arquitectura de la computadora. La instrucción de bits **and** toma dos palabras como entrada y devuelve una palabra en la que cada bit es la conjunción lógica de los bits de igual posición de las palabras de entrada. Lo que es importante observar es que una sola instrucción de bits **and** puede calcular la intersección de 32 o de 64 bits *a la vez*.

Si una relación tuviese un millón de registros, cada mapa de bits contendría un millón de bits, o, lo que es equivalente, 128 kilobytes. Sólo se necesitan 31.250 instrucciones para calcular la intersección de dos mapas de bits de la relación, suponiendo un tamaño de palabra de 32 bits. Por tanto, el cálculo de intersecciones de mapas de bits es una operación extremadamente rápida.

Al igual que la intersección de mapas de bits resulta útil para calcular la conjunción de dos condiciones, la unión de mapas de bits resulta útil para calcular la disyunción de dos condiciones. El procedimiento para la unión de mapas de bits es exactamente igual que el de la intersección, salvo que se utiliza la instrucción de bits **or** en lugar de **and**.

La operación complemento se puede usar para calcular un predicado que incluya la negación de una condición, como **not** (*nivel-ingresos = L<sub>1</sub>*). El complemento de un mapa de bits se genera complementando cada uno de sus bits (el complemento de 1 es 0 y el complemento de 0 es 1). Puede parecer que **not** (*nivel\_ingresos = L<sub>1</sub>*) se puede implementar simplemente calculando el complemento del mapa de bits del nivel de ingresos  $L_1$ . Sin embargo, si se ha borrado algún registro, el mero cálculo del complemento del mapa de bits no resulta suficiente. Los bits correspondientes a esos registros serán 0 en el mapa de bits original, pero pasarán a ser 1 en el complemento, aunque el registro no exista. También surge un problema similar cuando el valor de un atributo es *nulo*. Por ejemplo, si el valor de *nivel\_ingresos* es nulo, el bit será 0 en el mapa de bits original para el valor  $L_1$  y 1 en el complementado.

Para asegurarse de que los bits correspondientes a registros borrados se definen como 0 en el resultado, hay que intersecar el mapa de bits complementado con el mapa de bits de existencia para desactivar los bits de los registros borrados. Análogamente, para manejar los valores nulos, también se debe intersecar el mapa de bits complementado con el complemento del mapa de bits para el valor *nulo*.<sup>1</sup>

1. El tratamiento de predicados como **is unknown** puede causar aún más complicaciones, que requerirían en general el uso de un mapa de bits adicional para determinar los resultados de las operaciones que son desconocidos.

Se puede contar rápidamente el número de bits que valen 1 en el mapa de bits si se emplea una técnica inteligente. Se puede mantener un array de 256 entradas, donde la entrada  $i$ -ésima almacene el número de bits que valen 1 en la representación binaria de  $i$ . Hay que definir el recuento inicial como 0. Se toma cada byte del mapa de bits, se usa para indexar en el array y se añade el recuento almacenado al recuento total. El número de operaciones de suma sería la octava parte del número de tuplas  $y$ , por tanto el proceso de recuento es muy eficiente. Un gran array (que emplee  $2^{16} = 65.536$  entradas), indexado por pares de bytes, dará incluso aceleraciones mayores, pero con un coste de almacenamiento mayor.

### 12.9.3 Mapas de bits y árboles B<sup>+</sup>

Los mapas de bits se pueden combinar con los índices normales de árboles B<sup>+</sup> para las relaciones donde unos pocos valores de los atributos sean extremadamente frecuentes y también aparezcan otros valores, pero con mucha menor frecuencia. En las hojas de los índices de los árboles B<sup>+</sup>, para cada valor se suele mantener una lista de todos los registros con ese valor para el atributo indexado. Cada elemento de la lista sería un identificador de registro que consta, al menos, de 32 bits, y normalmente más. Para cada valor que aparece en muchos registros se almacena un mapa de bits en lugar de una lista de registros.

Supóngase que un valor dado  $v_i$  aparece en la dieciseisava parte de los registros de una relación. Sea  $N$  el número de registros de la relación y supóngase que cada registro tiene un número de 64 bits que lo identifica. El mapa de bits sólo necesita sólo un bit por registro, o  $N$  en total. En cambio, la representación de lista necesita sesenta y cuatro bits por registro en el que aparezca el valor, o  $64 * N/16 = 4N$  bits. Por tanto, es preferible el mapa de bits para representar la lista de registros del valor  $v_i$ . En el ejemplo (con un identificador de registros de 64 bits), si menos de uno de cada sesenta y cuatro registros tiene un valor dado, es preferible la representación de lista de registros para la identificación de los registros con ese valor, ya que emplea menos bits que la representación con mapas de bits. Si más de uno de cada sesenta y cuatro registros tiene ese valor dado, es preferible la representación de mapas de bits.

Por tanto, los mapas de bits se pueden emplear como mecanismo de almacenamiento comprimido en los nodos hoja de los árboles B<sup>+</sup> de los valores que aparecen muy frecuentemente.

## 12.10 Definición de índices en SQL

La norma SQL no proporciona al usuario o administrador de la base de datos ninguna manera de controlar qué índices se crean y se mantienen por el sistema de base de datos. Los índices no se necesitan para la corrección, ya que son estructuras de datos redundantes. Sin embargo, los índices son importantes para el procesamiento eficiente de las transacciones, incluyendo las transacciones de actualización y consulta. Los índices son también importantes para un cumplimiento eficiente de las ligaduras de integridad. Por ejemplo, las implementaciones típicas obligan a declarar una clave (Capítulo 4) mediante la creación de un índice con la clave declarada como la clave de búsqueda del índice.

En principio, un sistema de base de datos puede decidir automáticamente qué índices crear. Sin embargo, debido al coste en espacio de los índices, así como el efecto de los índices en el procesamiento de actualizaciones, no es fácil hacer una elección apropiada automáticamente sobre qué índices mantener. Por este motivo, la mayoría de las implementaciones de SQL proporcionan al programador control sobre la creación y eliminación de índices mediante comandos del lenguaje de definición de datos.

A continuación se ilustrará las sintaxis de estos comandos. Aunque la sintaxis que se muestra se usa ampliamente y está soportada en muchos sistemas de bases de datos, no es parte de la norma SQL:1999. Las normas SQL (hasta SQL:1999, al menos) no dan soporte al control del esquema físico de la base de datos y este libro se limita al esquema lógico de la base de datos.

Un índice se crea mediante el comando **create index**, que tiene la forma

```
create index <nombre-índice> on <nombre-relación> (<lista-atributos>)
```

*lista-atributos* es la lista de atributos de la relación que constituye la clave de búsqueda del índice.

Para definir un índice llamado *índice\_sucursal* de la relación *sucursal* con la clave de búsqueda *nombre\_sucursal*, se escribe

```
create index índice_sucursal on sucursal (nombre_sucursal)
```

Si se desea declarar que la clave de búsqueda es una clave candidata hay que añadir el atributo **unique** a la definición del índice. Con esto, el comando

```
create unique index índice_sucursal on sucursal (nombre_sucursal)
```

declara *nombre\_sucursal* como una clave candidata de *sucursal*. Si cuando se introduce el comando **create unique index**, *nombre\_sucursal* no es una clave candidata, se mostrará un mensaje de error y el intento de crear un índice fallará. Por otro lado, si el intento de crear el índice ha tenido éxito, cualquier intento de insertar una tupla que viole la declaración de clave fallará. Hay que observar que el carácter **unique** es redundante si el sistema de bases de datos soporta la declaración **unique** de SQL estándar.

Muchos sistemas de bases de datos ofrecen un modo de especificar el tipo de índice que se va a utilizar (como los árboles B<sup>+</sup> o la asociación). Algunos sistemas de bases de datos permiten también que se declare uno de los índices de una relación como agrupado; el sistema almacena entonces la relación ordenada de acuerdo con la clave de búsqueda del índice agrupado.

Para hacer posible la eliminación (drop) de un índice es necesario especificar su nombre. El comando **drop index** tiene la forma:

```
drop index <nombre Índice>
```

## 12.11 Resumen

- Muchas consultas solamente hacen referencia a una pequeña proporción de los registros de un archivo. Para reducir el gasto adicional en la búsqueda de estos registros se pueden construir *índices* para los archivos almacenados en la base de datos.
- Los archivos secuenciales indexados son unos de los esquemas de índice más antiguos usados en los sistemas de bases de datos. Para permitir una rápida recuperación de los registros según el orden de la clave de búsqueda, los registros se almacenan consecutivamente y los que no siguen el orden se encadenan entre sí. Para permitir un acceso aleatorio, se usan estructuras índice.
- Existen dos tipos de índices que se pueden utilizar: los índices densos y los índices dispersos. Los índices densos contienen una entrada por cada valor de la clave de búsqueda, mientras que los índices dispersos contienen entradas sólo para algunos de esos valores.
- Si el orden de una clave de búsqueda se corresponde con el orden secuencial del archivo, un índice sobre la clave de búsqueda se conoce como *índice con agrupación*. Los otros índices son los *índices sin agrupación* o *secundarios*. Los índices secundarios mejoran el rendimiento de las consultas que utilizan otras claves de búsqueda distinta de la del índice con agrupación. Sin embargo, éstas implican un gasto adicional en la modificación de la base de datos.
- El inconveniente principal de la organización del archivo secuencial indexado es que el rendimiento disminuye según crece el archivo. Para superar esta deficiencia se puede usar un *índice de árbol B<sup>+</sup>*.
- Un índice de árbol B<sup>+</sup> tiene la forma de un árbol *equilibrado*, en el cual cada camino de la raíz a las hojas del árbol tiene la misma longitud. La altura de un árbol B<sup>+</sup> es proporcional al logaritmo en base *N* del número de registros de la relación, donde cada nodo interno almacena *N* punteros; el valor de *N* está usualmente entre 50 y 100. Los árboles B<sup>+</sup> son más cortos que otras estructuras de árboles binarios equilibrados como los árboles AVL y, por tanto, necesitan menos accesos a disco para localizar los registros.
- Las búsquedas en un índice de árbol B<sup>+</sup> son directas y eficientes. Sin embargo, la inserción y el borrado son algo más complicados pero eficientes. El número de operaciones que se necesitan para la inserción y borrado en un árbol B<sup>+</sup> es proporcional al logaritmo en base *N* del número de registros de la relación, donde cada nodo interno almacena *N* punteros.
- Se pueden utilizar los árboles B<sup>+</sup> tanto para indexar un archivo con registros, como para organizar los registros de un archivo.

- Los índices de árbol B son similares a los índices de árbol B<sup>+</sup>. La mayor ventaja de un árbol B es que el árbol B elimina el almacenamiento redundante de los valores de la clave de búsqueda. Los inconvenientes principales son la complejidad y el reducido grado de salida para un tamaño de nodo dado. En la práctica, los índices de árbol B<sup>+</sup> están universalmente mejor considerados que los índices de árbol B por los diseñadores de sistemas.
- Las organizaciones de archivos secuenciales necesitan una estructura de índice para localizar los datos. Los archivos con organizaciones basadas en asociación, en cambio, permiten encontrar la dirección de un elemento de datos directamente mediante el cálculo de una función con el valor de la clave de búsqueda del registro deseado. Ya que no se sabe a la hora de diseñar la manera precisa en la cual los valores de la clave de búsqueda se van a almacenar en el archivo, una buena función de asociación a elegir es la que distribuya los valores de la clave de búsqueda a los cajones de una manera uniforme y aleatoria.
- La *asociación estática* utiliza una función de asociación en la que el conjunto de direcciones de cajones está fijado. Estas funciones de asociación no se pueden adaptar fácilmente a las bases de datos que tengan un crecimiento significativo con el tiempo. Hay varias *técnicas de asociación dinámica* que permiten que la función de asociación cambie. Un ejemplo es la *asociación extensible*, que trata los cambios de tamaño de la base datos mediante la división y fusión de cajones según crezca o disminuya la base de datos.
- También se puede utilizar la asociación para crear índices secundarios; tales índices se llaman *índices asociativos*. Por motivos de notación se asume que las organizaciones de archivos asociativos tienen un índice asociativo implícito en la clave de búsqueda usada para la asociación.
- Los índices ordenados con árboles B<sup>+</sup> y con índices asociativos se pueden usar para la selección basada en condiciones de igualdad que involucren varios atributos. Cuando hay varios atributos en una condición de selección se pueden interseccar los identificadores de los registros recuperados con los diferentes índices.
- Los índices de mapas de bits proporcionan una representación muy compacta para la indexación de atributos con muy pocos valores distintos. Las operaciones de intersección son extremadamente rápidas en los mapas de bits, haciéndolos ideales para el soporte de consultas con varios atributos.

## Términos de repaso

- Tipos de acceso.
- Tiempo de acceso.
- Tiempo de inserción.
- Tiempo de borrado.
- Espacio adicional.
- Índice ordenado.
- Índice con agrupación.
- Índice primario.
- Índice sin agrupación.
- Índice secundario.
- Archivo secuencial indexado.
- Registro/entrada del índice.
- Índice denso.
- Índice disperso.
- Índice multinivel.
- Clave compuesta.
- Exploración secuencial.
- Índice de árbol B<sup>+</sup>.
- Árbol equilibrado.
- Organización de archivos con árboles B<sup>+</sup>.
- Índice de árbol B.
- Asociación estática.
- Organización de archivos asociativos.
- Índice asociativo.
- Cajón.
- Función de asociación.
- Desbordamiento de cajones.
- Atasco.
- Asociación cerrada.
- Asociación dinámica.
- Asociación extensible.

- Acceso bajo varias claves.
  - Índices sobre varias claves.
  - Índice de mapas de bits.
  - Operaciones de mapas de bits:
- Intersección.
  - Unión.
  - Complemento.
  - Mapa de bits de existencia.

## Ejercicios prácticos

- 12.1 Dado que los índices agilizan el procesamiento de consultas, ¿por qué no deberían de mantenerse en varias claves de búsqueda? Enumérense tantas razones como sea posible.
- 12.2 ¿Es posible en general tener dos índices con agrupación en la misma relación para dos claves de búsqueda diferentes? Razónese la respuesta.
- 12.3 Constrúyase un árbol B<sup>+</sup> con el siguiente conjunto de valores de la clave:

$$(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)$$

Supóngase que el árbol está inicialmente vacío y que se añaden los valores en orden ascendente. Constrúyanse árboles B<sup>+</sup> para los casos en los que el número de punteros que caben en un nodo son:

- a. Cuatro
- b. Seis
- c. Ocho

- 12.4 Para cada árbol B<sup>+</sup> del Ejercicio práctico 12.3 muéstrese el aspecto del árbol después de cada una de las siguientes operaciones:
- a. Insertar 9.
  - b. Insertar 10.
  - c. Insertar 8.
  - d. Borrar 23.
  - e. Borrar 19.

- 12.5 Considérese el esquema modificado de redistribución para árboles B<sup>+</sup> descrito en la página 414. ¿Cuál es la altura esperada del árbol en función de  $n$ ?

- 12.6 Repítase el Ejercicio práctico 12.3 para un árbol B.

- 12.7 Supóngase que se está usando la asociación extensible en un archivo que contiene registros con los siguientes valores de la clave de búsqueda:

$$2, 3, 5, 7, 11, 17, 19, 23, 29, 31$$

Muéstrese la estructura asociativa extensible para este archivo si la función de asociación es  $h(x) = x \bmod 8$  y los cajones pueden contener hasta tres registros.

- 12.8 Muéstrese cómo cambia la estructura asociativa extensible del Ejercicio práctico 12.7 como resultado de realizar los siguientes pasos:
- a. Borrar 11.
  - b. Borrar 31.
  - c. Insertar 1.
  - d. Insertar 15.

- 12.9 Dese un pseudocódigo para el borrado de entradas de una estructura asociativa extensible, incluyendo detalles del momento y forma de fusionar cajones. No se debe considerar la reducción del tamaño de la tabla de direcciones de cajones.

- 12.10 Sugiérase una forma eficaz de comprobar si la tabla de direcciones de cajones en una asociación extensible se puede reducir en tamaño almacenando un recuento extra con la tabla de direcciones

de cajones. Dense detalles de cómo se debería mantener el recuento cuando se dividen, fusionan o borran los cajones.

*Nota:* la reducción del tamaño de la tabla de direcciones de cajones es una operación costosa y las inserciones subsecuentes pueden causar que la tabla vuelva a crecer. Por tanto, es mejor no reducir el tamaño tan pronto como se pueda, sino solamente si el número de entradas de índice es pequeño en comparación con el tamaño de la tabla de direcciones de los cajones.

**12.11** Considérese la relación *cuenta* mostrada en la Figura 12.25.

- Constrúyase un índice de mapa de bits sobre los atributos *nombre\_sucursal* y *saldo*, dividiendo *saldo* en cuatro rangos: menores que 250, entre 250 y menor que 500, entre 500 y menor que 750, y 750 o mayor.
  - Considérese una consulta que solicite todas las cuentas de Daimiel con un saldo de 500 o más. Describanse los pasos para responder a la consulta y muéstrense los mapas de bits finales e intermedios construidos para responder la consulta.
- 12.12** Supóngase que se tiene una relación con  $n_r$  tuplas sobre la que se va a construir un índice secundario de árbol  $B^+$ .
- Dese una fórmula para el coste de construir el índice de árbol  $B^+$  insertando un registro cada vez. Supóngase que cada página contendrá en media  $f$  entradas, y que todos los niveles del árbol sobre la raíz están en memoria.
  - Suponiendo un tiempo de acceso a disco de 10 milisegundos, ¿cuál es el coste de la construcción del índice para una relación con 10 millones de registros?
  - Sugírase una manera *ascendente* más eficiente de crear el índice, construyendo en primer lugar todo el nivel de hojas y luego los niveles superiores uno a uno. Supóngase que se tiene una función que puede ordenar de manera eficiente un conjunto de registros muy grande, aunque sea tan grande que no cabe en memoria. (Estos algoritmos de ordenación se describen más adelante, en el Apartado 13.4 y, suponiendo una cantidad razonable de memoria principal tienen un coste de alrededor de una operación de E/S por bloque.)

## Ejercicios

- 12.13** ¿Cuándo es preferible utilizar un índice denso en vez de un índice disperso? Razónese la respuesta.
- 12.14** ¿Cuál es la diferencia entre un índice con agrupación y un índice secundario?
- 12.15** Para cada árbol  $B^+$  del Ejercicio práctico 12.3 muéstrense los pasos involucrados en las siguientes consultas:
- Encontrar los registros con un valor de la clave de búsqueda de 11.
  - Encontrar los registros con un valor de la clave de búsqueda entre 7 y 17, ambos inclusive.
- 12.16** La solución presentada en el Apartado 12.5.3 para tratar las claves de búsqueda duplicadas añadían un atributo adicional a la clave de búsqueda. ¿Qué efecto tiene este cambio en la altura del árbol  $B^+$ ?
- 12.17** Explíquense las diferencias entre la asociación abierta y la cerrada. Coméntense los beneficios de cada técnica en aplicaciones de bases de datos.
- 12.18** ¿Cuáles son las causas del desbordamiento de cajones en un archivo con una organización asociativa? ¿Qué se puede hacer para reducir la aparición del desbordamiento de cajones?
- 12.19** ¿Por qué una estructura asociativa no es la mejor elección para una clave de búsqueda en la que son probables las consultas de rangos?
- 12.20** Supóngase la relación  $R(A, B, C)$ , con un índice de árbol  $B^+$  con la clave de búsqueda  $(A, B)$ .
- ¿Cuál es el máximo coste posible de buscar registros que satisfagan  $10 < A < 50$  al usar este índice en términos del número de registros recuperados  $n_1$  y la altura  $h$  del árbol?

- b. ¿Cuál es el máximo coste posible de buscar registros que satisfagan  $10 < A < 50 \wedge 5 < B < 10$  al usar este índice en términos del número de registros recuperados  $n_2$ , y de  $n_1$  y de  $h$ , como se definieron antes?
- c. ¿Bajo qué condiciones sobre  $n_1$  y sobre  $n_2$  el índice sería una forma eficiente de buscar los registros que satisfagan  $10 < A < 50 \wedge 5 < B < 10$ ?
- 12.21 Supóngase que hay que crear un índice de árbol B<sup>+</sup> sobre un gran número de nombres, donde el tamaño máximo de un nombre puede ser grande (por ejemplo, 40 caracteres) y el tamaño medio también (10 caracteres). Explíquese cómo se puede usar la compresión de prefijo para maximizar el grado de salida medio de los nodos internos.
- 12.22 ¿Por qué podrían perder secuencialidad los nodos hoja de una organización de archivos con árboles B<sup>+</sup>? Sugiérase cómo se podría reorganizar para restaurar la secuencialidad.
- 12.23 Supóngase una relación almacenada en una organización de archivo de árbol B+. Supóngase que los índices secundarios con identificadores de registro que son punteros a los registros del disco.
- ¿Cuál sería el efecto sobre los índices secundarios si ocurre una división de página en la organización de archivo?
  - ¿Cuál sería el coste de la actualización de todos los registros afectados en un índice secundario?
  - ¿Cómo soluciona este problema el uso de una clave de búsqueda como un identificador lógico de registro?
  - ¿Cuál es el coste adicional debido al uso de estos identificadores lógicos?
- 12.24 Muéstrese la forma de calcular mapas de existencia a partir de otros mapas de bits. Asegúrese de que la técnica funciona incluso con valores nulos, usando un mapa de bits para el valor *nulo*.
- 12.25 ¿Cómo afecta el cifrado de datos a los esquemas de índices? En particular, ¿cómo afectaría a los esquemas que intentan almacenar los datos de manera ordenada?
- 12.26 Nuestra descripción de la asociación estática asume que una gran extensión contigua de bloques de disco se pueden ubicar en una tabla asociativa estática. Supóngase que se pueden asignar sólo  $C$  bloques contiguos. Sugiérase cómo implementar la tabla asociativa si puede ser mucho más grande que los  $C$  bloques. El acceso a bloque debería seguir siendo eficiente.

## Notas bibliográficas

En Cormen et al. [1990] se pueden encontrar explicaciones acerca de las estructuras básicas utilizadas en la indexación y asociación. Los índices de árbol B se introdujeron por primera vez en Bayer [1972] y en Bayer y McCreight [1972]. Los árboles B<sup>+</sup> se tratan en Comer [1979], Bayer y Unterauer [1977] y Knuth [1973]. Las notas bibliográficas del Capítulo 16 proporcionan referencias a la investigación sobre los accesos concurrentes y las actualizaciones en los árboles B<sup>+</sup>. Gray y Reuter [1993] proporciona una buena descripción de los resultados en la implementación de árboles B<sup>+</sup>.

Se han propuesto varias estructuras alternativas de árboles y basadas en árboles. Los tries son unos árboles cuya estructura está basada en los “dígitos” de las claves (por ejemplo, el índice de muescas de un diccionario, con una entrada para cada letra). Estos árboles podrían no estar equilibrados en el sentido que lo están los árboles B<sup>+</sup>. Los tries se estudian en Ramesh et al. [1989], Orenstein [1982], Litwin [1981] y Fredkin [1960]. Otros trabajos relacionados son los árboles B digitales de Lomet [1981]. Knuth [1973] analiza un gran número de técnicas de asociación distintas. Existen varias técnicas de asociación dinámica. Fagin et al. [1979] introducen la asociación extensible. La asociación lineal se introduce en Litwin [1978] y Litwin [1980]; en Rathi et al. [1990] se presentó una comparación de rendimiento con la asociación extensible. Una alternativa propuesta en Ramakrishna y Larson [1989] permite la recuperación en un solo acceso a disco al precio de una gran sobrecarga en una pequeña fracción de las modificaciones de la base de datos. La asociación dividida es una extensión de la asociación para varios atributos, y se trata en Rivest [1976], Burkhard [1976] y Burkhard [1979].

Vitter [2001] proporciona una extensa visión general de las estructuras de datos en memoria externa y algoritmos para ellas.

Los índices de mapas de bits y las variantes denominadas **índices por capas de bits e índices de proyección** se describen en O'Neil y Quass [1997]. Se introdujeron por primera vez en el gestor de archivos Model 204 de IBM sobre la plataforma AS 400. Proporcionan grandes ganancias de velocidad en ciertos tipos de consultas y se encuentran implementadas actualmente en la mayoría de sistemas de bases de datos. Entre la investigación más reciente en índices de mapas de bits figura la de Wu y Buchmann [1998], Chan y Ioannidis [1998], Chan y Ioannidis [1999] y Johnson [1999].



# Procesamiento de consultas

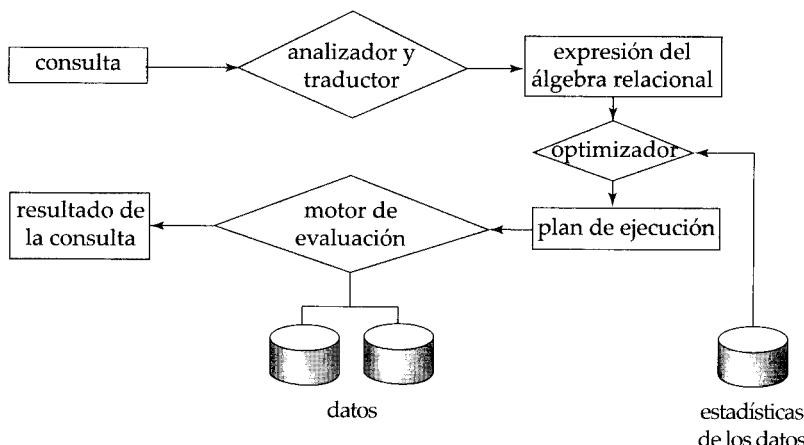
El **procesamiento de consultas** hace referencia a una serie de actividades implicadas en la extracción de datos de una base de datos. Estas actividades incluyen la traducción de consultas expresadas en lenguajes de bases de datos de alto nivel en expresiones implementadas en el nivel físico del sistema, así como transformaciones de optimización de consultas y la evaluación real de las mismas.

## 13.1 Visión general

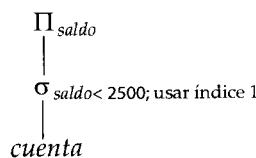
En la Figura 13.1 se ilustran los pasos involucrados en el procesamiento de una consulta. Los pasos básicos son:

1. Análisis y traducción.
2. Optimización.
3. Evaluación.

Antes de empezar el procesamiento de una consulta, el sistema debe traducirla a una forma utilizable. Un lenguaje como SQL es adecuado para el uso humano, pero es poco apropiado para una representación interna en el sistema de la consulta. Una representación interna más útil estaría basada en el álgebra relacional extendido.



**Figura 13.1** Pasos en el procesamiento de una consulta.



**Figura 13.2** Plan de ejecución de una consulta.

Así, la primera acción que el sistema tiene que emprender para procesar una consulta es su traducción a su formato interno. Este proceso de traducción es similar al trabajo que realiza el analizador de un compilador. Durante la generación del formato interno de una consulta, el analizador comprueba la sintaxis de la consulta del usuario, verifica que los nombres de las relaciones que aparecen en ella sean nombres de relaciones en la base de datos, etc. Posteriormente se construye un árbol para el análisis de la consulta, que se transformará en una expresión del álgebra relacional. Si la consulta estuviera expresada en términos de una vista, la fase de traducción también sustituye todas las referencias a vistas por las expresiones del álgebra relacional que las definen<sup>1</sup>. El análisis de lenguajes se describe en la mayoría de los libros sobre compiladores (véanse las notas bibliográficas).

Dada una consulta hay generalmente distintos métodos para obtener la respuesta. Por ejemplo, ya se ha visto que en SQL se puede expresar una consulta de diferentes maneras. Cada consulta SQL puede traducirse a diferentes expresiones del álgebra relacional. Además de esto, la representación de una consulta en el álgebra relacional especifica de manera parcial cómo evaluar la consulta; existen normalmente varias maneras de evaluar expresiones del álgebra relacional. Como ejemplo, considérese la consulta:

```

select saldo
from cuenta
where saldo < 2500

```

Esta consulta se puede traducir en alguna de las siguientes expresiones del álgebra relacional:

- $\sigma_{saldo < 2500} (\Pi_{saldo} (cuenta))$
- $\Pi_{saldo} (\sigma_{saldo < 2500} (cuenta))$

Además, se puede ejecutar cada operación del álgebra relacional utilizando alguno de los diferentes algoritmos. Por ejemplo, para implementar la selección anterior se puede examinar cada tupla de *cuenta* para encontrar las tuplas cuyo saldo sea menor que 2500. Por otro lado, si se dispone de un índice de árbol  $B^+$  en el atributo *saldo*, se puede utilizar este índice para localizar las tuplas.

Para especificar completamente cómo evaluar una consulta, no basta con proporcionar la expresión del álgebra relacional, además hay que anotar en ella las instrucciones que especifiquen cómo evaluar cada operación. Estas anotaciones podrían ser el algoritmo a usar para una operación específica o el índice o índices concretos a utilizar. Las operaciones del álgebra relacional anotadas con instrucciones sobre su evaluación reciben el nombre de **primitivas de evaluación**. La secuencia de operaciones primitivas que se pueden utilizar en la evaluación de una consulta establece un **plan de ejecución de la consulta** o un **plan de evaluación de la consulta**. En la Figura 13.2 se ilustra un plan de evaluación para nuestro ejemplo de consulta, en el que se especifica un índice concreto (denotado en la figura como “índice 1”) para la operación selección. El **motor de ejecución de consultas** escoge un plan de evaluación, lo ejecuta y devuelve su respuesta a la consulta.

Los diferentes planes de evaluación de una misma consulta dada pueden tener costes distintos. No se puede esperar que los usuarios escriban las consultas de manera que sugieran el plan de evaluación más eficiente. En su lugar, es responsabilidad del sistema construir un plan de evaluación de la consulta

1. Para vistas materializadas, la expresión que define la vista ha sido ya evaluada y almacenada. Por tanto, se puede usar la relación almacenada en lugar de reemplazar los usos de la vista por la expresión que define la vista. Las vistas recursivas se tratan de manera diferente mediante un procedimiento de búsqueda de punto fijo, según se vio en los Apartados 4.7 y 5.4.6.

que minimice el coste de la evaluación de la consulta; esta tarea se denomina *optimización de consultas*. El Capítulo 14 describe en detalle la optimización de consultas.

Una vez elegido el plan de la consulta, ésta se evalúa con este plan y se muestra su resultado.

La secuencia de pasos que se han descrito para procesar una consulta son representativos; no todas las bases de datos los siguen exactamente. Por ejemplo, en lugar de utilizar la representación del álgebra relacional, varias bases de datos usan una representación del árbol de análisis con anotaciones basada en la estructura de la consulta SQL. Sin embargo, los conceptos que se describen aquí constituyen la base del procesamiento de consultas en las bases de datos.

Para optimizar una consulta, el optimizador de consultas debe conocer el coste de cada operación. Aunque el coste exacto es difícil de calcular, dado que depende de muchos parámetros como la memoria real disponible, es posible obtener una estimación aproximada del coste de ejecución para cada operación.

En este capítulo se estudia la forma de evaluar operaciones individuales en un plan de consulta y cómo estimar su coste; se vuelve a la optimización de consultas en el Capítulo 14. En el Apartado 13.2 se describe cómo se mide el coste de una consulta. Desde el Apartado 13.3 hasta el 13.6 se estudia la evaluación de operaciones individuales del álgebra relacional. Varias operaciones se pueden agrupar en un **cauce**, en el que cada una de las ellas empieza trabajando sobre sus tuplas de entrada del mismo modo que si fuesen generadas por otra operación. En el Apartado 13.7 se examina cómo coordinar la ejecución de varias operaciones de un plan de evaluación de consultas, en particular, cómo usar las operaciones encauzadas para evitar escribir resultados intermedios en disco.

## 13.2 Medidas del coste de una consulta

El coste de la evaluación de una consulta se puede expresar en términos de diferentes recursos, incluyendo los accesos a disco, el tiempo de CPU en ejecutar una consulta y, en sistemas de bases de datos distribuidos o paralelos, el coste de la comunicación (que se estudiará más tarde en los Capítulos 21 y 22). El tiempo de respuesta para un plan de evaluación de una consulta (esto es, el tiempo de reloj que se necesita para ejecutar el plan), si se supone que no hay otra actividad ejecutándose en el sistema, podría tener en cuenta todos estos costes y utilizarlos como una buena medida del coste del plan.

Sin embargo, en grandes sistemas de bases de datos, el coste de acceso a los datos en disco es normalmente el más importante, ya que los accesos a disco son más lentos comparados con las operaciones en memoria. Además, la velocidad de la CPU aumenta mucho más rápidamente que las velocidades de los discos. Así, lo más probable es que el tiempo empleado en operaciones del disco siga influyendo en el tiempo total de ejecución de una consulta. Las estimaciones del tiempo de CPU son más difíciles de hacer porque dependen de detalles de bajo nivel del código de ejecución. Aunque los optimizadores reales de consultas consideran los costes de CPU, aquí se ignoran y sólo se considera el coste de los accesos a disco para medir el coste del plan de evaluación de una consulta.

Se usarán el *número de transferencias de bloques* de disco y el *número de búsquedas de disco* para medir el coste del acceso a datos en disco. Si al subsistema de disco le lleva una media de  $t_T$  segundos para transferir un bloque de datos, con un tiempo medio de acceso a bloque (tiempo de búsqueda en el disco más latencia rotacional) de  $t_S$  segundos, entonces una operación que transfiera  $b$  bloques y execute  $S$  búsquedas tardaría  $b * t_T + S * t_S$  segundos. Los valores de  $t_T$  y de  $t_S$  se deben ajustar al disco usado, pero los valores normales de los discos de altas prestaciones actuales serían  $t_S = 4$  milisegundos y  $t_T = 0.1$  milisegundos, asumiendo un tamaño de bloque de 4 kilobytes y una velocidad de transferencia de 40 megabytes por segundo<sup>2</sup>.

Se puede refinar aún más la estimación de coste distinguiendo entre las lecturas y escrituras de bloques, dado que normalmente se tarda el doble de tiempo en escribir un bloque que en leerlo de disco (debido a que los sistemas de disco leen los sectores después de escribirlos para comprobar que la escritura fue correcta). Para simplificar se ignorará este detalle y se propone al lector que desarrolle estimaciones de coste más precisas para distintas operaciones.

Las estimaciones de coste que se proporcionan no incluyen el coste de escribir el resultado final de una operación en disco. Esto se tendrá en cuenta cuando sea preciso. Los costes de todos los algoritmos

2. Algunos sistemas de bases de datos ejecutan búsquedas y transferencias de bloque de prueba para estimar los costes medios de búsqueda y transferencia de bloque, como parte del proceso de instalación del software.

que se consideran aquí dependen del tamaño de la memoria intermedia en la memoria principal. En el mejor caso, todos los datos se pueden leer en las memorias intermedias y no es necesario acceder de nuevo a disco. En el peor caso se asume que la memoria intermedia puede contener sólo unos pocos bloques de datos (aproximadamente un bloque por relación). Al presentar las estimaciones de coste se asume generalmente el peor caso.

Además, aunque se asuma que los datos se deben leer inicialmente de disco, es posible que el bloque al que se acceda pueda estar en la memoria intermedia. De nuevo y para simplificar se ignorará este efecto; como resultado el coste del acceso a disco real durante la ejecución de un plan puede ser menor que el coste estimado.

## 13.3 Operación selección

En el procesamiento de consultas, el **explorador de archivo** es el operador de nivel más bajo para acceder a los datos. Los exploradores de archivo son algoritmos de búsqueda que localizan y recuperan los registros que cumplen una condición de selección. En los sistemas relacionales, el explorador de archivo permite leer una relación completa en los casos en que la relación se almacena en un único archivo dedicado.

### 13.3.1 Algoritmos básicos

Considérese una operación selección en una relación cuyas tuplas se almacenan juntas en un archivo. A continuación se muestran dos algoritmos exploradores que implementan la operación selección:

- **A1 (búsqueda lineal).** En una búsqueda lineal se explora cada bloque del archivo y se comprueban todos los registros para determinar si satisfacen o no la condición de selección. Para una selección sobre un atributo clave, el sistema puede terminar la exploración si encuentra el registro requerido sin necesidad de examinar los otros registros de la relación.

El coste de la búsqueda lineal, en términos de operaciones E/S, es una búsqueda más  $b_r$  transferencias de bloque, donde  $b_r$  denota el número de bloques del archivo o, de forma equivalente, el coste temporal es  $t_S + b_r * t_T$

En una selección sobre un atributo clave el sistema puede terminar la exploración si se encuentra el registro requerido, sin examinar los otros registros de la relación. Las selecciones sobre atributos clave presentan un coste medio de  $b_r/2$ , pero en el peor caso aún tiene un coste de  $b_r$  transferencias de bloque más una búsqueda.

Aunque podría ser más lento que otros algoritmos para la implementación de la selección, el algoritmo de búsqueda lineal se puede aplicar a cualquier archivo, sin importar su ordenación, la presencia de índices o la naturaleza de la operación selección. El resto de algoritmos que se estudiarán no son aplicables en todos los casos, pero cuando lo son, son más rápidos que la búsqueda lineal.

- **A2 (búsqueda binaria).** Si el archivo está ordenado según un atributo y la condición de la selección es una comparación de igualdad en ese atributo, se puede utilizar una búsqueda binaria para localizar los registros que satisfacen la selección. El sistema realiza la búsqueda binaria de los bloques del archivo.

En el peor caso, el número de bloques que deben ser examinados para encontrar los registros requeridos es  $\lceil \log_2(b_r) \rceil$ , donde  $b_r$  denota el número de bloques del archivo. Cada uno de estos accesos a bloque requiere una búsqueda además de una transferencia de bloque, y por tanto el coste temporal es  $\lceil \log_2(b_r) \rceil * (t_T + t_S)$ .

Si la selección es sobre un atributo que no sea clave, es posible que más de un bloque contenga los registros requeridos, y el coste de la lectura de los bloques extra se debe añadir a la estimación del coste. Se puede calcular este número mediante la estimación del tamaño del resultado de la selección (que se explica en el Apartado 14.3) y dividiéndolo entre el número de registros que se almacenan por bloque de la relación. Se asume que estos bloques se almacenan contiguos por lo que sólo se añade un coste de transferencia  $t_T$  por cada bloque adicional.

### 13.3.2 Selecciones con índices

Las estructuras índice se denominan **caminos de acceso**, ya que proporcionan un camino a través del cual se pueden localizar y acceder a los datos. En el Capítulo 12 se señaló la eficiencia de leer los registros del archivo en un orden próximo al orden físico. Recuérdese que un *índice primario* (también conocido como *índice con agrupación*) es un índice que permite leer los registros de un archivo en el mismo orden que el orden físico del archivo. Un índice que no es primario se llama *índice secundario*.

Los algoritmos de búsqueda que utilizan un índice reciben el nombre de **exploraciones del índice**. Los índices ordenados, como los árboles  $B^+$ , también permiten acceder a las tuplas según cierto orden que es útil para la implementación de las consultas de rangos. Aunque los índices pueden proporcionar un acceso rápido, directo y ordenado, su uso implica un gasto adicional en los accesos a los bloques que contienen el índice. Utilizaremos el predicado de selección como guía en la elección del índice a usar en el procesamiento de la consulta. Los algoritmos de búsqueda que usan un índice son:

- **A3 (índice primario, igualdad basada en la clave).** Para una condición de igualdad en un atributo clave con un índice primario se puede utilizar el índice para recuperar el único registro que satisface la correspondiente condición de igualdad.

Si se usa un árbol  $B^+$ , el coste de la operación en términos de operaciones E/S es igual a la altura del árbol más una operación E/S para recuperar el registro<sup>3</sup>; cada una de las cuales requiere una búsqueda y una transferencia de bloque. Por tanto, el coste es  $(h_i + 1) * (t_T + t_S)$ , donde  $h_i$  denota la altura del índice<sup>4</sup>.

- **A4 (índice primario, igualdad basada en un atributo no clave).** Se pueden recuperar varios registros mediante el uso de un índice primario cuando la condición de selección especifica una comparación de igualdad en un atributo  $A$  que no sea clave. La única diferencia del caso anterior es que puede ser necesario recuperar varios registros. Sin embargo, estos registros estarían almacenados consecutivamente en el archivo ya que el archivo se ordena según la clave de búsqueda.

El coste de la operación depende de la altura del árbol más el número de bloques que contengan registros con la clave de búsqueda especificada. Se necesita una búsqueda para cada nivel del árbol. Además se necesita una búsqueda para obtener el primer bloque que contenga el registro deseado; el resto de bloques se almacenan consecutivamente y no requieren más búsquedas. En concreto el coste es  $h_i * (t_T + t_S) + t_S + b * t_T$ , donde  $h_i$  denota la altura del índice y  $b$  denota el número de bloques que contienen bloques con la clave de búsqueda especificada.

- **A5 (índice secundario, igualdad).** Las selecciones con una condición de igualdad pueden utilizar un índice secundario. Esta estrategia puede recuperar un único registro si la condición de igualdad es sobre una clave; puede que se recuperen varios registros si el campo índice no es clave.

En el primer caso sólo se obtiene un registro, y el coste es igual a la altura del árbol más una operación E/S para recuperar el registro. Cada operación E/S requiere una búsqueda y una transferencia de bloque. El coste temporal en este caso es el mismo que el del índice primario (caso A3).

En el segundo caso, cada registro puede residir en un bloque diferente, que puede provocar una operación E/S por cada registro recuperado, y cada operación E/S requiere una búsqueda y una transferencia de bloque. El coste podría llegar a ser incluso peor que el de la búsqueda lineal si se obtiene un gran número de registros. El coste temporal es en este caso  $(h_i + n) * (t_S + t_T)$ , donde  $n$  es el número de registros recuperados<sup>5</sup>.

3. Si se usa un árbol  $B^+$ , no se requiere E/S adicional porque los registros se almacenan en las hojas del árbol. Se deberían hacer ajustes similares a algunos de los algoritmos descritos a continuación en este apartado cuando se usen árboles  $B^+$ .

4. Los optimizadores reales asumen generalmente que la raíz del árbol está en la memoria intermedia porque se accede frecuentemente a ella. Algunos optimizadores incluso asumen que todos menos las hojas se encuentran en memoria porque se accede a ellos con relativa frecuencia, y normalmente el uno por ciento de los nodos no son hojas. La fórmula del coste se debe modificar adecuadamente.

5. Si la memoria intermedia es grande, el bloque que contenga el registro puede estar ya en memoria principal. Es posible construir una estimación del coste medio o esperado de la selección teniendo en cuenta la probabilidad de que el bloque se encuentre ya en memoria. Para grandes memorias intermedias la estimación será mucho menor que la estimación del peor caso.

Como se describió en el Apartado 12.5.5, cuando los registros se almacenan en una organización de árbol B<sup>+</sup> u otras organizaciones que puedan requerir la reubicación de los registros, los índices secundarios generalmente no almacenan punteros a los registros<sup>6</sup>. En su lugar, estos índices almacenan los valores de los atributos usados como la clave de búsqueda en una organización de árbol B<sup>+</sup>. El acceso a un registro mediante un índice secundario es por tanto más caro: primero se busca en el índice secundario para encontrar los valores de la clave de búsqueda del índice primario y después se busca en el índice primario para encontrar los registros. La fórmula del coste descrita para índices secundarios se tendrá que modificar adecuadamente si se usan estos índices.

### 13.3.3 Selecciones con condiciones de comparación

Considérese una selección de la forma  $\sigma_{A \leq v}(r)$ . Se pueden implementar utilizando búsqueda lineal o binaria, o con índices de alguna de las siguientes maneras:

- **A6 (índice primario, comparación).** Se puede utilizar un índice ordenado primario (por ejemplo, un índice primario de árbol B<sup>+</sup>) cuando la condición de selección sea una comparación. Para condiciones con comparaciones de la forma  $A > v$  o  $A \geq v$  se puede usar el índice primario sobre A para guiar la recuperación de las tuplas de la manera siguiente. Para el caso de  $A \geq v$  se busca el valor de  $v$  en el índice para encontrar la primera tupla del archivo que tenga un valor de  $A = v$ . Un explorador de archivo comenzando en esa tupla y hasta el final del archivo devuelve todas las tuplas que satisfacen la condición. Para  $A > v$ , el explorador de archivo comienza con la primera tupla tal que  $A > v$ . La estimación de coste en este caso es idéntica a la del caso A4.

Para comparaciones de la forma  $A < v$  o  $A \leq v$  no es necesario buscar en el índice. Para el caso de  $A < v$  se utiliza un simple explorador partiendo del inicio del archivo y continuando hasta (pero sin incluirlo) la primera tupla con el atributo  $A = v$ . El caso  $A \leq v$  es similar, excepto que el explorador continúa hasta (pero sin incluir) la primera tupla con el atributo  $A > v$ . En ninguno de los dos casos el índice es de utilidad alguna.

- **A7 (índice secundario, comparación).** Se puede utilizar un índice secundario ordenado para guiar la recuperación bajo condiciones de comparación que contengan  $<$ ,  $\leq$ ,  $\geq$ , o  $>$ . Los bloques del índice del nivel más bajo se exploran o bien desde el valor más pequeño hasta  $v$  (para  $<$  y  $\leq$ ) o bien desde  $v$  hasta el valor mayor (para  $>$  y  $\geq$ ).

El índice secundario proporciona punteros a los registros, pero para obtener los reales es necesario extraerlos usando los punteros. Este paso puede requerir una operación E/S por cada registro extraído, dado que los consecutivos pueden estar en diferentes bloques de disco; como antes, cada operación E/S requiere una búsqueda y una transferencia de bloque. Si el número de registros extraídos es grande, el uso del índice secundario puede ser incluso más caro que la búsqueda lineal. Por tanto, el índice secundario sólo se debería usar si se seleccionan muy pocos registros.

### 13.3.4 Implementación de selecciones complejas

Hasta ahora sólo se han considerado condiciones de selección simples de la forma  $A op B$ , donde  $op$  es una operación igualdad o de comparación. Se revisarán a continuación predicados de selección más complejos.

- **Conjunción.** Una *selección conjuntiva* es una selección de la forma

$$\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$$

- **Disyunción.** Una *selección disyuntiva* es una selección de la forma

$$\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$$

6. Recuérdese que si se usan organizaciones de árboles B<sup>+</sup> para almacenar relaciones, los registros se pueden trasladar entre bloques cuando los nodos hoja se dividen o fusionan, y cuando se redistribuyen los registros.

Una condición disyuntiva se cumple mediante la unión de todos los registros que cumplen alguna de las condiciones  $\theta_i$ .

- **Negación.** El resultado de una selección  $\sigma_{\neg\theta}(r)$  es el conjunto de tuplas de  $r$  para las que la condición  $\theta$  es falsa. A falta de valores nulos, este conjunto es simplemente el conjunto de tuplas que no están en  $\sigma_\theta(r)$ .

Se puede implementar una operación selección con una conjunción o una disyunción de condiciones sencillas utilizando alguno de los siguientes algoritmos:

- **A8 (selección conjuntiva utilizando un índice).** Inicialmente hay que determinar si para un atributo hay disponible algún camino de acceso en alguna de las condiciones simples. Si lo hay, cualquiera de los algoritmos de selección A2 hasta A7 puede recuperar los registros que cumplan esa condición. Se completa la operación mediante la comprobación en la memoria intermedia de que cada registro recuperado cumpla o no el resto de condiciones simples.

Para reducir el coste, se elige un  $\theta_i$  y uno de los algoritmos entre A1 y A7 para el que la combinación resulte en el menor coste de  $\sigma_{\theta_i}(r)$ . El coste del algoritmo A8 está determinado por el coste del algoritmo elegido.

- **A9 (selección conjuntiva utilizando un índice compuesto).** Puede que se disponga de un *índice compuesto* (es decir, un índice sobre varios atributos) apropiado para algunas selecciones conjuntivas. Si la selección especifica una condición de igualdad en dos o más atributos y existe un índice compuesto en estos campos con atributos combinados, entonces se podría buscar en el índice directamente. El tipo de índice determina cuál de los algoritmos A3, A4 o A5 se utilizará.

- **A10 (selección conjuntiva mediante la intersección de identificadores).** Otra alternativa para implementar la operación selección conjuntiva implica la utilización de punteros a registros o identificadores de registros. Este algoritmo necesita índices con punteros a registros en los campos involucrados por cada condición individual. De este modo se explora cada índice en busca de punteros cuyas tuplas cumplan alguna condición individual. La intersección de todos los punteros recuperados forma el conjunto de punteros a tuplas que satisfacen la condición conjuntiva. Luego se usa el conjunto de punteros para recuperar los registros reales. Si no hubiera índices disponibles para algunas condiciones concretas entonces habría que comprobar el resto de condiciones de los registros recuperados.

El coste del algoritmo A10 es la suma de los costes de las cada una de las exploraciones del índice más el coste de recuperar los registros en la intersección de las listas recuperadas de punteros. Este coste se puede reducir ordenando la lista de punteros y recuperando registros en orden. Por tanto, (1) todos los punteros a los registros de un bloque van juntos, y así todos los registros seleccionados en el bloque se pueden recuperar usando una única operación E/S, y (2) los bloques se leen ordenados, minimizando el movimiento del brazo del disco. El Apartado 13.4 describe algunos algoritmos de ordenación.

- **A11 (selección disyuntiva mediante la unión de identificadores).** Si se dispone de caminos de acceso en todas las condiciones de la selección disyuntiva, se explora cada índice en busca de punteros cuyas tuplas cumplan una condición individual. La unión de todos los punteros recuperados proporciona el conjunto de punteros a todas las tuplas que cumplen la condición disyuntiva. Despues se utilizan estos punteros para recuperar los registros reales.

Sin embargo, aunque sólo una de las condiciones no tenga un camino de acceso, se tiene que realizar una búsqueda lineal en la relación para encontrar todas las tuplas que cumplen la condición. Por tanto, aunque sólo exista una de estas condiciones en la disyunción, el método de acceso más eficiente es una exploración lineal, que comprueba durante la exploración la condición disyuntiva en cada tupla.

La implementación de selecciones con condiciones negativas se deja como ejercicio (Ejercicio práctico 13.6).

## 13.4 Ordenación

La ordenación de los datos juega un papel importante en los sistemas de bases de datos por dos razones. Primero, las consultas SQL pueden solicitar que los resultados sean ordenados. Segundo, e igualmente importante para el procesamiento de consultas, varias de las operaciones relacionales, como las reuniones, se pueden implementar eficientemente si las relaciones de entrada están ordenadas. Por este motivo se revisa la ordenación antes que la operación reunión en el Apartado 13.5.

La ordenación se puede conseguir mediante la construcción de un índice en la clave de ordenación y utilizando luego ese índice para leer la relación de manera ordenada. No obstante, este proceso ordena la relación sólo *lógicamente* a través de un índice, en lugar de *físicamente*. Por tanto, la lectura de las tuplas de manera ordenada podría implicar un acceso a disco (búsqueda más transferencia de bloque) para cada tupla, lo que puede ser muy costoso dado que el número de registros puede ser mucho mayor que el número de bloques. Por esta razón sería deseable ordenar las tuplas físicamente.

El problema de la ordenación se ha estudiado ampliamente para los casos en que la relación cabe completamente en memoria principal, y para el caso en el que la relación es mayor que la memoria. En el primer caso se utilizan técnicas de ordenación clásicas como la ordenación rápida (*quick-sort*). Aquí se estudia cómo tratar el segundo caso.

La ordenación de relaciones que no caben en memoria se denomina **ordenación externa**. La técnica más utilizada para la ordenación externa es normalmente el algoritmo de **ordenación-mezcla externa**. A continuación se describe el algoritmo de ordenación-mezcla externa. Sea  $M$  el número de marcos de página en la memoria intermedia de la memoria principal (el número de bloques de disco cuyos contenidos se pueden alojar en la memoria intermedia de la memoria principal).

1. En la primera etapa, se crean varias **secuencias** ordenadas; cada **secuencia** está ordenada, pero sólo contiene parte de los registros de la relación.

```

 $i = 0;$
repeat
 leer M bloques, o bien de la relación, o bien del resto de la relación,
 según el que tenga menor número de bloques;
 ordenar la parte de la relación que está en la memoria;
 escribir los datos ordenados al archivo de secuencias S_i ;
 $i = i + 1$;
until el final de la relación

```

2. En la segunda etapa, las secuencias se *mezclan*. Supóngase que, por ahora, el número total de secuencias  $N$  es menor que  $M$ , así que se puede asignar un marco de página para cada secuencia y reservar espacio para guardar una página con el resultado. La etapa de mezcla se lleva a cabo de la siguiente manera:

```

leer un bloque de cada uno de los N archivos S_i y
guardarlos en una página de la memoria intermedia en memoria;
repeat
 elegir la primera tupla (según el orden) de entre todas las páginas
 de la memoria intermedia;
 escribir la tupla y suprimirla de la página de la memoria intermedia;
 if la página de la memoria intermedia de alguna secuencia S_i está vacía
 and not fin-de-archivo(S_i)
 then leer el siguiente bloque de S_i y guardarlo en la página de la memoria intermedia;
 until todas las páginas de la memoria intermedia estén vacías

```

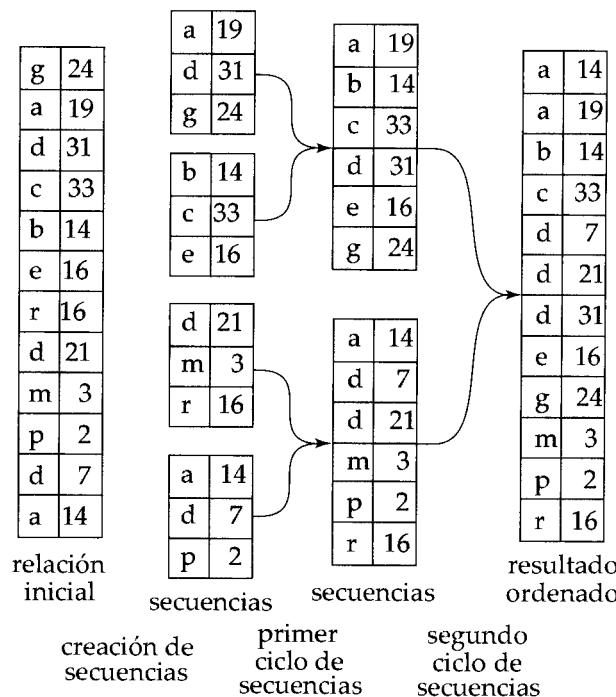
El resultado de la etapa de mezcla es la relación ya ordenada. El archivo de salida se almacena en una memoria intermedia para reducir el número de operaciones de escritura en el disco. La operación anterior de mezcla es una generalización de la mezcla de dos vías utilizado por el algoritmo normal de ordenación-mezcla en memoria; éste mezcla  $N$  secuencias, por eso se llama **mezcla de  $N$  vías**.

En general, si la relación es mucho mayor que la memoria, se podrían generar  $M$  o más secuencias en la primera etapa y no sería posible asignar un marco de página para cada secuencia durante la etapa de mezcla. En este caso, se realiza la operación de mezcla en varios ciclos. Como hay suficiente memoria para  $M - 1$  páginas de la memoria intermedia de entrada, cada mezcla puede tomar  $M - 1$  secuencias como entrada.

El *ciclo* inicial se realiza como sigue. Se mezclan las  $M - 1$  secuencias primeras (según se ha descrito ya en el punto 2) para obtener una única secuencia en el siguiente ciclo. Luego se mezclan de manera similar las siguientes  $M - 1$  secuencias, continuando así hasta que todas las secuencias iniciales se hayan procesado. En este punto, el número de secuencias se ha reducido por un factor de  $M - 1$ . Si el número reducido de secuencias es todavía mayor o igual que  $M$ , se realiza otro ciclo con las secuencias creadas en el primer ciclo como entrada. Cada ciclo reduce el número de secuencias por un factor de  $M - 1$ . Estos ciclos se repiten tantas veces como sea necesario, hasta que el número de secuencias sea menor que  $M$ ; momento en el que un último ciclo genera el resultado ordenado.

En la Figura 13.3 se ilustran los pasos de la ordenación-mezcla externa en una relación ficticia. Por motivos didácticos supóngase que solamente cabe una tupla en cada bloque ( $f_r = 1$ ) y que la memoria puede contener como mucho tres marcos de página. Durante la etapa de mezcla se utilizan dos marcos de página como entrada y uno para la salida.

El coste de acceso a disco de la ordenación-mezcla externa se calcula de la siguiente manera: sea  $b_r$  el número de bloques que contienen registros de la relación  $r$ . En la primera etapa, se lee y se copia de nuevo cada bloque de la relación, lo que resulta en un total de  $2b_r$  transferencias de bloques. El número inicial de secuencias es  $\lceil b_r/M \rceil$ . Puesto que el número de secuencias decrece en un factor de  $M - 1$  en cada ciclo de la mezcla, el número total de ciclos requeridos viene dado por la expresión  $\lceil \log_{M-1}(b_r/M) \rceil$ . Cada uno de estos ciclos lee y copia todos los bloques de la relación una vez, con dos excepciones. En primer lugar, el ciclo final puede producir la ordenación como resultado sin escribir el resultado en disco. En segundo lugar, puede que haya secuencias que ni lean ni copien durante un ciclo (por ejemplo, si hay  $M$  secuencias que mezclar en un ciclo, se leen y se mezclan  $M - 1$ , y no se accede a una de las secuencias durante ese ciclo). Si se ignora el ahorro (relativamente pequeño) debido a este



**Figura 13.3** Ordenación externa utilizando ordenación-mezcla.

último efecto, el número total de transferencias de bloques para la ordenación externa de la relación es

$$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$$

Aplicando esta ecuación al ejemplo de la Figura 13.3 se obtiene un total de  $12 * (4+1) = 60$  transferencias de bloques, como se puede comprobar en la figura. Obsérvese que este número no incluye el coste de escribir el resultado final.

También es necesario añadir los costes de la búsqueda en disco. La generación de ciclos requiere búsquedas para la lectura de datos para cada una de las secuencias, así como para escribir las secuencias. Durante la fase de mezcla, si se leen simultáneamente  $b_b$  bloques de cada secuencia (es decir, se asignan  $b_b$  bloques de memoria intermedia a cada secuencia), entonces cada paso de mezcla requeriría cerca de  $\lceil b_r/b_b \rceil$  búsquedas para la lectura de los datos<sup>7</sup>. Aunque la salida se escriba secuencialmente, si está en el mismo disco que las secuencias de entrada, la cabeza se puede haber desplazado entre las escrituras de los bloques consecutivos. Así que habría que añadir un total de  $2\lceil b_r/b_b \rceil$  búsquedas por cada paso de mezcla, excepto el paso final (ya que se asume que el resultado final no se escribe a disco). El número total de búsquedas es, por tanto:

$$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil(2\lceil \log_{M-1}(b_r/M) \rceil - 1)$$

Aplicando esta ecuación al ejemplo de la Figura 13.3 se obtiene un total de  $8 + 12 * (2 * 2 - 1) = 44$  búsquedas si se establece el número de bloques por secuencia  $b_b$  como 1.

## 13.5 Operación reunión

En este apartado se estudian varios algoritmos para calcular la reunión de relaciones y para analizar sus costes asociados.

Se utiliza la palabra **equirreunión** para hacer referencia a las reuniones de la forma  $r \bowtie_{r.A=s.B} s$ , donde  $A$  y  $B$  son atributos o conjuntos de atributos de las relaciones  $r$  y  $s$ , respectivamente.

Utilizaremos como ejemplo la expresión

$$\text{impositor} \bowtie \text{cliente}$$

la cual utiliza los mismos esquemas de relación de los Capítulos 2 y 3. Se da por supuesta la siguiente información de catálogo acerca de las dos relaciones:

- Número de registros de *cliente*:  $n_{\text{cliente}} = 10.000$ .
- Número de bloques de *cliente*:  $b_{\text{cliente}} = 400$ .
- Número de registros de *impositor*:  $n_{\text{impositor}} = 5.000$ .
- Número de bloques de *impositor*:  $b_{\text{impositor}} = 100$ .

### 13.5.1 Reunión en bucle anidado

En la Figura 13.4 se muestra un algoritmo sencillo para calcular la reunión zeta,  $r \bowtie_\theta s$ , de dos relaciones  $r$  y  $s$ . Este algoritmo se llama de **reunión en bucle anidado**, ya que básicamente consiste en un par de bucles **for** anidados. La relación  $r$  se denomina la **relación externa** y  $s$  la **relación interna** de la reunión, puesto que el bucle de  $r$  incluye al bucle de  $s$ . El algoritmo utiliza la notación  $t_r \cdot t_s$ , donde  $t_r$  y  $t_s$  son tuplas;  $t_r \cdot t_s$  denota la tupla construida mediante la concatenación de los valores de los atributos de las tuplas  $t_r$  y  $t_s$ .

Al igual que el algoritmo de búsqueda lineal en archivos de la selección, el algoritmo de reunión en bucle anidado tampoco necesita índices y se puede utilizar sin importar la condición de la reunión. La manera de extender el algoritmo para calcular la reunión natural es directa, puesto que la reunión natural se puede expresar como una reunión zeta seguida de la eliminación de los atributos repetidos mediante una proyección. El único cambio que se necesita es un paso adicional para borrar los atributos repetidos de la tupla  $t_r \cdot t_s$  antes de añadirla al resultado.

7. Para ser más precisos, dado que cada secuencia se lee por separado y se pueden obtener menos de  $b_b$  bloques al leer el final de una secuencia, es posible que se necesite una búsqueda adicional por cada secuencia. Para simplificar se ignorará este detalle.

```

for each tupla t_r in r do begin
 for each tupla t_s in s do begin
 comprobar que el par (t_r, t_s) satisface la condición θ de la reunión
 si la cumple, añadir $t_r \cdot t_s$ al resultado.
 end
end

```

**Figura 13.4** Reunión en bucle anidado.

```

for each bloque B_r of r do begin
 for each bloque B_s of s do begin
 for each tupla t_r in B_r do begin
 for each tupla t_s in B_s do begin
 comprobar que el par (t_r, t_s) satisface la condición de la reunión
 si la cumple, añadir $t_r \cdot t_s$ al resultado.
 end
 end
 end
end

```

**Figura 13.5** Reunión en bucle anidado por bloques.

El algoritmo de reunión en bucle anidado es costoso, ya que examina cada pareja de tuplas en las dos relaciones. Considérese el coste del algoritmo de reunión en bucle anidado. El número de pares de tuplas a considerar es  $n_r * n_s$ , donde  $n_r$  denota el número de tuplas en  $r$ , y  $n_s$  denota el número de tuplas en  $s$ . Para registro de  $r$  se tiene que realizar una exploración completa de  $s$ . En el peor caso, la memoria intermedia solamente puede contener un bloque de cada relación, necesitándose un total de  $n_r * b_s + b_r$  transferencias de bloques, donde  $b_r$  y  $b_s$  denotan el número de bloques que contienen tuplas de  $r$  y de  $s$ , respectivamente. Sólo se necesita una búsqueda por cada exploración de la relación interna  $s$ , dado que se lee secuencialmente, y un total de  $b_r$  búsquedas para leer  $r$ , por lo que resulta un total de  $n_r + b_r$  búsquedas. En el mejor caso hay suficiente espacio para que las dos relaciones quepan en memoria, así que cada bloque se tendrá que leer solamente una vez; en consecuencia, sólo se necesitarán  $b_r + b_s$  transferencias de bloques, además de 2 búsquedas.

Si una de las relaciones cabe en memoria por completo, es útil usar esa relación como la relación más interna, ya que solamente será necesario leer una vez la relación del bucle más interno. Por lo tanto, si  $s$  es lo suficientemente pequeña para caber en la memoria principal, esta estrategia necesita solamente un total de  $b_r + b_s$  transferencias de bloques y dos búsquedas (el mismo coste que en el caso en que las dos relaciones quepan en memoria).

Considérese ahora el caso de la reunión natural de *impositor* y *cliente*. Supóngase que, por el momento, no hay ningún índice en cualquiera de las relaciones y que no se desea crear ninguno. Se pueden utilizar los bucles anidados para calcular la reunión; supóngase que *impositor* es la relación externa y que *cliente* es la relación interna de la reunión. Se tendrán que examinar  $5.000 * 10.000 = 50 * 10^6$  pares de tuplas. En el peor de los casos el número de transferencias de bloques será de  $5.000 * 400 + 100 = 2.000.100$ , más  $5.000 + 100 = 5.100$  búsquedas. En la mejor de las situaciones, sin embargo, se tienen que leer ambas relaciones solamente una vez y realizar el cálculo. Este cálculo necesita a lo sumo  $100 + 400 = 500$  transferencias de bloques más dos búsquedas (una mejora significativa respecto de la peor situación posible). Si se hubiera utilizado *cliente* como la relación del bucle externo e *impositor* para el bucle interno, el coste en el peor de los casos de la última estrategia habría sido menor:  $10.000 * 100 + 400 = 1.000.400$  transferencias de bloques más 10.400 búsquedas. El número de transferencias de bloques es significativamente menor y, aunque el número de búsquedas es superior, el coste total se reduce, suponiendo que  $t_S = 4$  milisegundos y que  $t_T = 0,1$  milisegundos.

### 13.5.2 Reunión en bucle anidado por bloques

Si la memoria intermedia es demasiado pequeña para contener las dos relaciones por completo en memoria, todavía se puede lograr un mayor ahorro en los accesos a los bloques si se procesan las relaciones por bloques en lugar de por tuplas. En la Figura 13.5 se muestra la **reunión en bucle anidado por bloques**, que es una variante de la reunión en bucle anidado donde se empareja cada bloque de la relación interna con cada bloque de la relación externa. En cada par de bloques se empareja cada tupla de un bloque con cada tupla del otro bloque para generar todos los pares de tuplas. Al igual que antes se añaden al resultado todas las parejas de tuplas que satisfacen la condición de la reunión.

La diferencia principal en coste entre la reunión en bucle anidado por bloques y la reunión en bucle anidado básica es que, en el peor de los casos, cada bloque de la relación interna  $s$  se lee solamente una vez por cada *bloque* de la relación externa, en lugar de una vez por cada *tupla* de la relación externa. De este modo, en el peor de los casos, habrá un total de  $b_r * b_s + b_r$  transferencias de bloques, donde  $b_r$  y  $b_s$  denotan respectivamente el número de bloques que contienen registros de  $r$  y de  $s$ , respectivamente. Cada exploración de la relación interna requiere una búsqueda y la exploración de la relación externa una búsqueda por bloque, dando un total de  $2 * b_r$  búsquedas. Evidentemente, será más eficiente utilizar la relación más pequeña como la relación externa, en caso de que ninguna de ellas quepa en memoria. En el mejor de los casos, cuando la relación interna quepa en memoria, habrá que realizar  $b_r + b_s$  transferencias de bloques y sólo dos búsquedas (en este caso se escogería la relación de menor tamaño como relación interna).

En el ejemplo del cálculo de  $impositor \bowtie cliente$  se usará ahora el algoritmo de reunión en bucle anidado por bloques. En el peor de los casos hay que leer una vez cada bloque de *cliente* por cada bloque de *impositor*. Así, en el peor caso, son necesarias un total de  $100 * 400 + 100 = 40.100$  transferencias de bloques más  $2 * 100 = 200$  búsquedas. Este coste supone una mejora importante frente a las  $5.000 * 400 + 100 = 2.000.100$  transferencias de bloques más 5.100 búsquedas necesarias en el peor de los casos para la reunión en bucle anidado básica. El coste en el mejor caso sigue siendo el mismo (es decir,  $100 + 400 = 500$  transferencias de bloques y dos búsquedas).

El rendimiento de los procedimientos de bucle anidado y bucle anidado por bloques se puede mejorar aún más:

- Si los atributos de la reunión en una reunión natural o en una equirreunión forman una clave de la relación interna, entonces el bucle interno puede finalizar tan pronto como se encuentre la primera correspondencia.
- En el algoritmo en bucle anidado por bloques, en lugar de utilizar bloques de disco como la unidad de bloqueo de la relación externa, se puede utilizar el mayor tamaño que quepa en memoria, mientras quede suficiente espacio para las memorias intermedias de la relación interna y la salida. En otras palabras, si la memoria tiene  $M$  bloques, se leen  $M - 2$  bloques de la relación externa de una vez, y cuando se lee cada bloque de la relación interna se reúne con los  $M - 2$  bloques de la relación externa. Este cambio reduce el número de exploraciones de la relación interna de  $b_r$  a  $\lceil b_r / (M - 2) \rceil$ , donde  $b_r$  es el número de bloques de la relación externa. El coste total es  $\lceil b_r / (M - 2) \rceil * b_s + b_r$  transferencias de bloques y  $2\lceil b_r / (M - 2) \rceil$  búsquedas.
- Se puede explorar el bucle interno alternativamente hacia adelante y hacia atrás. Este método de búsqueda ordena las peticiones de bloques de disco de tal manera que los datos restantes en la memoria intermedia de la búsqueda anterior se reutilizan, reduciendo de este modo el número de accesos a disco necesarios.
- Si se dispone de un índice en un atributo de la reunión del bucle interno se pueden sustituir las búsquedas en archivos por búsquedas más eficientes en el índice. Esta optimización se describe en el Apartado 13.5.3.

### 13.5.3 Reunión en bucle anidado indexada

En una reunión en bucle anidado (Figura 13.4), si se dispone de un índice sobre el atributo de la reunión del bucle interno, se pueden sustituir las exploraciones de archivo por búsquedas en el índice. Para cada

tupla  $t_r$  de la relación externa  $r$ , se utiliza el índice para buscar tuplas en  $s$  que cumplan la condición de reunión con la tupla  $t_r$ .

Este método de reunión se denomina **reunión en bucle anidado indexada**; se puede utilizar cuando existen índices y cuando se crean índices temporales con el único propósito de evaluar la reunión.

La búsqueda de tuplas en  $s$  que cumplan las condiciones de la reunión con una tupla dada  $t_r$  es esencialmente una selección en  $s$ . Por ejemplo, considérese  $impositor \bowtie cliente$ . Supóngase que se tiene una tupla de  $impositor$  con  $nombre\_cliente$  "Martín". Entonces, las tuplas relevantes de  $s$  son aquellas que satisfacen la selección " $nombre\_cliente = Martín$ ".

El coste de la reunión en bucle anidado indexada se puede calcular como se indica a continuación. Para cada tupla en la relación externa  $r$  se realiza una búsqueda en el índice para  $s$  recuperando las tuplas apropiadas. En el peor de los casos sólo hay espacio en la memoria intermedia para una página de  $r$  y una página del índice. Por tanto, son necesarios  $b_r$  operaciones de E/S para leer la relación  $r$ , donde  $b_r$  denota el número de bloques que contienen registros de  $r$ ; cada E/S requiere una búsqueda y una transferencia de bloque ya que la cabeza del disco se puede haber trasladado entre cada E/S. Para cada tupla de  $r$ , se realiza una búsqueda en el índice para  $s$ . Por tanto, el coste temporal de la reunión se puede calcular como  $b_r(t_T + t_S) + n_r * c$ , donde  $n_r$  es el número de registros de la relación  $r$  y  $c$  es el coste de una única selección en  $s$  utilizando la condición de la reunión. Ya se vio en el Apartado 13.3 cómo estimar el coste del algoritmo de una única selección (posiblemente utilizando índices) cuyo cálculo proporciona el valor de  $c$ .

La fórmula del coste indica que, si hay índices disponibles en ambas relaciones  $r$  y  $s$ , normalmente es más eficiente usar como relación más externa aquella que tenga menos tuplas.

Por ejemplo, considérese una reunión en bucle anidado indexada de  $impositor \bowtie cliente$ , con  $impositor$  como relación externa. Supóngase también que  $cliente$  tiene un índice de árbol  $B^+$  primario en el atributo de la reunión  $nombre\_cliente$ , que contiene 20 entradas en promedio por cada nodo del índice. Dado que  $cliente$  tiene 10.000 tuplas, la altura del árbol es 4, y será necesario un acceso más para encontrar el dato real. Como  $n_{impositor}$  es 5.000, el coste total es  $100 + 5000 * 5 = 25.100$  accesos a disco, cada uno de los cuales requiere una búsqueda y una transferencia de bloque. En cambio, como se indicó anteriormente, se necesitan 40.100 transferencias de bloque más 200 búsquedas para una reunión en bucle anidado por bloques. Aunque se ha reducido el número de transferencias de bloques, el coste de búsqueda se ha incrementado, aumentando el coste total dado que una búsqueda es considerablemente más costosa que una transferencia de bloque. Sin embargo, si se tuviese una selección sobre la relación  $impositor$  que redujera significativamente el número de filas, la reunión en bucle anidado sería significativamente más rápida que una reunión en bucle anidado por bloques.

### 13.5.4 Reunión por mezcla

El algoritmo de **reunión por mezcla** (también llamado algoritmo de **reunión por ordenación-mezcla**) se puede utilizar para calcular reuniones naturales y equirreuniones. Sean  $r(R)$  y  $s(S)$  las relaciones que vamos a utilizar para realizar la reunión natural, y sean  $R \cap S$  sus atributos en común. Supóngase que ambas relaciones están ordenadas según los atributos de  $R \cap S$ . Por tanto, su reunión se puede calcular mediante un proceso muy parecido a la etapa de mezcla del algoritmo de ordenación-mezcla.

El algoritmo de reunión por mezcla se muestra en la Figura 13.6. En este algoritmo, *AtribsReunión* se refiere a los atributos de  $R \cap S$  y, a su vez,  $t_r \bowtie t_s$ , donde  $t_r$  y  $t_s$  son las tuplas que tienen los mismos valores en *AtribsReunión*, y denota la concatenación de los atributos de las tuplas, seguido de una proyección para no incluir los atributos repetidos. El algoritmo de reunión por mezcla asocia un puntero con cada relación. Al comienzo, estos punteros apuntan a la primera tupla de sus respectivas relaciones. Según avanza el algoritmo, el puntero se mueve a través de la relación. De este modo se leen en  $S_s$  un grupo de tuplas de una relación con el mismo valor en los atributos de la reunión. El algoritmo de la Figura 13.6 *necesita* que cada conjunto de tuplas  $S_s$  quepa en memoria principal; más adelante, en este apartado, se examinarán extensiones del algoritmo que evitan este supuesto. Después, las tuplas correspondientes de la otra relación (si las hay) se leen y se procesan según se están leyendo.

En la Figura 13.7 se muestran dos relaciones que están ordenadas en su atributo de la reunión *a1*. Es instructivo examinar los pasos del algoritmo de reunión por mezcla con las relaciones que se muestran en la figura.

Dado que las relaciones están ordenadas, las tuplas con el mismo valor en los atributos de la reunión aparecerán consecutivamente. De este modo solamente es necesario leer ordenadamente cada tupla una vez y, como resultado, leer también cada bloque solamente una vez. Puesto que sólo se recorre un ciclo en ambos archivos, el método de reunión por mezcla resulta eficiente; el número de transferencias de bloques es igual a la suma de los bloques en los dos archivos,  $b_r + b_s$ .

Asumiendo que se asignen  $b_b$  bloques de memoria intermedia para cada relación, el número de búsquedas requeridas sería  $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ . Puesto que las búsquedas son mucho más costosas que las transferencias de bloques, tiene sentido asignar varios bloques de la memoria intermedia para cada relación, siempre que haya memoria disponible. Por ejemplo con  $t_T = 0.1$  milisegundos por bloque de 4 megabytes y  $t_S = 4$  milisegundos, el tamaño de la memoria intermedia es de 400 bloques (o 1,6 megabytes), el tiempo de búsqueda sería 4 milisegundos por cada 40 milisegundos de tiempo de transferencia, en otras palabras, el tiempo de búsqueda sería sólo el 10 por ciento del tiempo de transferencia.

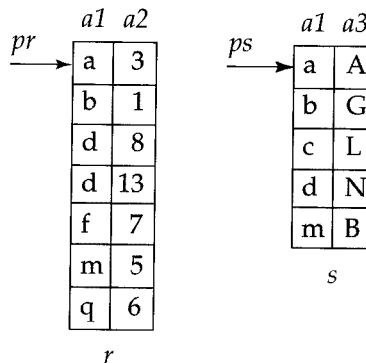
Si alguna de las relaciones de entrada  $r$  o  $s$  no está ordenada según los atributos de la reunión, se pueden ordenar primero y luego utilizar el algoritmo de reunión por mezcla. El algoritmo de reunión por mezcla también se puede extender fácilmente desde las reuniones naturales al caso más general de las equirreuniones.

```

 $pr :=$ dirección de la primera tupla de r ;
 $ps :=$ dirección de la primera tupla de s ;
while ($ps \neq \text{null}$ and $pr \neq \text{null}$) do
 begin
 $t_s :=$ tupla a la que apunta ps ;
 $S_s := \{t_s\}$;
 hacer que ps apunte a la siguiente tupla de s ;
 $hecho := \text{falso}$;
 while (not $hecho$ and $ps \neq \text{null}$) do
 begin
 $t_s' :=$ tupla a la que apunta ps ;
 if ($t_s'[AtribsReunión] = t_s[AtribsReunión]$)
 then begin
 $S_s := S_s \cup \{t_s'\}$;
 hacer que ps apunte a la siguiente tupla de s ;
 end
 else $hecho := \text{cierto}$;
 end
 $t_r :=$ tupla a la que apunta pr ;
 while ($pr \neq \text{null}$ and $t_r[AtribsReunión] < t_s[AtribsReunión]$) do
 begin
 hacer que pr apunte a la siguiente tupla de r ;
 $t_r :=$ tupla a la que apunta pr ;
 end
 while ($pr \neq \text{null}$ and $t_r[AtribsReunión] = t_s[AtribsReunión]$) do
 begin
 for each t_s in S_s do
 begin
 añadir $t_s \bowtie t_r$ al resultado;
 end
 hacer que pr apunte a la siguiente tupla de r ;
 $t_r :=$ tupla a la que apunta pr ;
 end
 end.

```

**Figura 13.6** Reunión por mezcla.



**Figura 13.7** Relaciones ordenadas para la reunión por mezcla.

Como se mencionó anteriormente, el algoritmo de reunión por mezcla de la Figura 13.6 requiere que el conjunto  $S_s$  de todas las tuplas con el mismo valor de los atributos de la reunión deben caber en memoria principal. Este requisito se suele cumplir, incluso cuando la relación  $s$  es grande. Si no se puede cumplir, se debería realizar una reunión en bucle anidado por bloques entre  $S_s$  y las tuplas de  $r$  con los mismos valores de los atributos de reunión. Por tanto, se incrementa el coste total de la reunión por mezcla.

También es posible realizar una variación de la operación reunión por mezcla en tuplas desordenadas si existen índices secundarios en los dos atributos de la reunión. Así, se examinan los registros a través de los índices recuperándolos de manera ordenada. Sin embargo, esta variación presenta un importante inconveniente, puesto que los registros podrían estar diseminados a través de los bloques del archivo. Por tanto, cada acceso a una tupla podría implicar acceder a un bloque del disco, y esto es muy costoso.

Para evitar este coste se podría utilizar una técnica de reunión por mezcla híbrida que combinase índices con reunión por mezcla. Supóngase que una de las relaciones se encuentra ordenada y la otra desordenada, pero con un índice secundario de árbol  $B^+$  en los atributos de la reunión. El **algoritmo híbrido de reunión por mezcla** fusiona la relación ordenada con las entradas hoja del índice secundario de árbol  $B^+$ . El archivo resultante contiene tuplas de la relación ordenada y direcciones de las tuplas de la relación desordenada. Después se ordena el archivo resultante según las direcciones de las tuplas de la relación desordenada, permitiendo así una recuperación eficiente de las correspondientes tuplas, según el orden físico de almacenamiento, para completar la reunión. Se deja como ejercicio la extensión de esta técnica para tratar el caso de dos relaciones desordenadas.

Supóngase que se aplica el esquema de reunión por mezcla al ejemplo de  $impositor \bowtie cliente$ . En este caso, el atributo de la reunión es *nombre\_cliente*. Supóngase además que las dos relaciones se encuentren ya ordenadas según el atributo de la reunión *nombre\_cliente*. En este caso, la reunión por mezcla emplea un total de  $400 + 100 = 500$  transferencias de bloques. Si se asume que en el peor caso sólo se asigna un bloque de memoria intermedia para cada relación de entrada (es decir,  $b_b = 1$ ), se necesitarían un total de  $400 + 100 = 500$  búsquedas; en realidad se puede hacer que  $b_b$  sea mucho mayor ya que se necesitan bloques de la memoria intermedia sólo para dos relaciones, y el coste de búsqueda sería significativamente menor.

Supóngase que las relaciones no están ordenadas y que el tamaño de memoria en el peor caso es de sólo tres bloques. El coste en este caso se calcularía:

1. Usando la fórmula desarrollada en el Apartado 13.4, la ordenación de la relación *cliente* hacen falta  $\lceil \log_{3-1}(400/3) \rceil = 8$  pasos de mezcla. La relación de esta relación necesitaría  $400 * (2\lceil \log_{3-1}(400/3) \rceil + 1)$ , o 6.800 transferencias de bloques, con otras 400 transferencias para escribir el resultado. El número de búsquedas necesarias es  $2 * \lceil 400/3 \rceil + 400 * (2 * 8 - 1)$ , 6.268 búsquedas para ordenar y 400 búsquedas para escribir el resultado, con un total de 6.668 búsquedas dado que sólo hay disponible un bloque de memoria intermedia para cada ciclo.
2. Análogamente, la ordenación de *impositor* necesita  $\lceil \log_{3-1}(100/3) \rceil = 6$  pasos de mezcla y  $100 * (2\lceil \log_{3-1}(100/3) \rceil + 1)$ , 1.300 transferencias de bloques, con otras 100 transferencias para escribir

el resultado. El número de búsquedas necesarias es  $2 * \lceil 100/3 \rceil + 100 * (2*6 - 1) = 1.164$  búsquedas para ordenar, y 100 búsquedas para escribir el resultado, con un total de 1.264 búsquedas.

3. Finalmente, la mezcla de ambas relaciones necesita  $400 + 100 = 500$  transferencias de bloques y 500 búsquedas.

Por tanto, el coste total es de 9.100 transferencias de bloques más 8.932 búsquedas si las relaciones no están ordenadas y el tamaño de memoria es sólo de 3 bloques.

Con un tamaño de memoria de 25 bloques y sin tener ordenadas las relaciones, el coste de la ordenación seguida de la reunión por mezcla sería:

1. La ordenación de *cliente* se puede hacer en un único paso de mezcla y requiere un total de  $400 * (2 \lceil \log_{24}(400/25) \rceil + 1) = 1.200$  transferencias de bloques. De forma similar, la ordenación de *impositor* emplea 300 transferencias de bloques. Escribir la salida ordenada a disco requiere  $400 + 100 = 500$  transferencias de bloques, y el paso de mezcla necesita 500 transferencias de bloques para volver a leer los datos. Al sumar estos costes se calcula un total de 2.500 transferencias de bloques.
2. Si se asume que sólo se asigna un bloque de memoria intermedia para cada secuencia, el número de búsquedas requeridas en este caso es  $2 * \lceil 400/25 \rceil + 400 + 400 = 832$  búsquedas para ordenar *cliente* y escribir la salida ordenada a disco y, análogamente,  $2 * \lceil 100/25 \rceil + 100 + 100 = 208$  para *impositor*, más  $400 + 100$  búsquedas para la lectura de los datos ordenados en el paso de reunión por mezcla. Al sumar estos costes se calcula un total de 1.640 búsquedas.

Se puede reducir significativamente el número de búsquedas asignando más bloques de memoria intermedia en cada ciclo. Por ejemplo, si se asignan cinco bloques y para la salida de la de la mezcla de las cuatro secuencias de *impositor*, el coste se reduce de 208 a  $2 * \lceil 100/25 \rceil + \lceil 100/5 \rceil + \lceil 100/5 \rceil = 48$  búsquedas. Si en el paso de reunión por mezcla se asignan 12 bloques para ambas relaciones, el número de búsquedas para este paso se reduce de 500 a  $\lceil 400/12 \rceil + \lceil 100/12 \rceil = 43$  búsquedas. Por tanto, el número total de búsquedas es 251.

Por tanto, el coste total es de 2.500 transferencias de bloques más 251 búsquedas si las relaciones no están ordenadas y el tamaño de memoria es de 25 bloques.

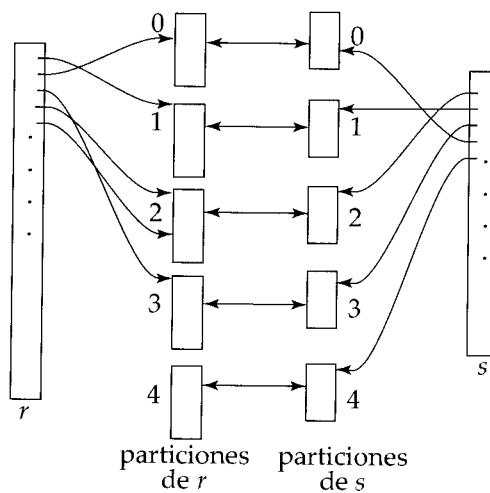
### 13.5.5 Reunión por asociación

Al igual que el algoritmo de reunión por mezcla, el algoritmo de reunión por asociación se puede utilizar para implementar reuniones naturales y equirreuniones. En el algoritmo de reunión por asociación se utiliza una función de asociación  $h$  para dividir las tuplas de ambas relaciones. La idea fundamental es dividir las tuplas de cada relación en conjuntos con el mismo valor de la función de asociación en los atributos de la reunión.

Se da por supuesto que

- $h$  es una función de asociación que asigna a los *AtribsReunión* los valores  $\{0, 1, \dots, n_h\}$ , donde los *AtribsReunión* denotan los atributos comunes de  $r$  y  $s$  utilizados en la reunión natural.
- $H_{r_0}, H_{r_1}, \dots, H_{r_{n_h}}$  denotan las particiones de las tuplas de  $r$ , inicialmente todas vacías. Cada tupla  $t_r \in r$  se pone en la partición  $H_{r_i}$ , donde  $i = h(t_r[AtribsReunión])$ .
- $H_{s_0}, H_{s_1}, \dots, H_{s_{n_h}}$  denotan las particiones de las tuplas de  $s$ , inicialmente todas vacías. Cada tupla  $t_s \in s$  se pone en la partición  $H_{s_i}$ , donde  $i = h(t_s[AtribsReunión])$ .

La función de asociación  $h$  debería de tener las “buenas” propiedades de aleatoriedad y uniformidad que se estudiaron en el Capítulo 12. La división de las relaciones se muestra de manera gráfica en la Figura 13.8.



**Figura 13.8** División por asociación de relaciones.

### 13.5.5.1 Fundamentos

La idea en que se basa el algoritmo de reunión por asociación es la siguiente. Supóngase que una tupla de  $r$  y una tupla de  $s$  satisfacen la condición de la reunión; por tanto, tendrán el mismo valor en los atributos de la reunión. Si el valor se asocia con algún valor  $i$ , la tupla de  $r$  tiene que estar en  $H_{r_i}$  y la tupla de  $s$  en  $H_{s_i}$ . De este modo solamente es necesario comparar las tuplas de  $r$  en  $H_{r_i}$  con las tuplas de  $s$  en  $H_{s_i}$ ; no es necesario compararlas con las tuplas de  $s$  de ninguna otra partición.

Por ejemplo, si  $i$  es una tupla de *impositor*,  $c$  una tupla de *cliente* y  $h$  una función de asociación en los atributos *nombre\_cliente* de las tuplas, solamente hay que comprobar  $i$  y  $c$  si  $h(c) = h(i)$ . Si  $h(c) \neq h(d)$ , entonces  $c$  e  $i$  poseen valores distintos de *nombre\_cliente*. Sin embargo, si  $h(c) = h(i)$  hay que comprobar que  $c$  e  $i$  tengan los mismos valores en los atributos de la reunión, ya que es posible que  $c$  e  $i$  tengan valores diferentes de *nombre\_cliente*, con el mismo valor de la función de asociación.

En la Figura 13.9 se muestran los detalles del algoritmo de **reunión por asociación** para calcular la reunión natural de las relaciones  $r$  y  $s$ . Como en el algoritmo de reunión por mezcla,  $t_r \bowtie t_s$  denota la concatenación de los atributos de las tuplas de  $t_r$  y  $t_s$ , seguido de la proyección para eliminar los atributos repetidos. Después de la división de las relaciones, el resto del código de reunión por asociación realiza una reunión en bucle anidado indexada separada en cada una de los particón pares  $i$ , con  $i = 0, \dots, n_h$ . Para lograr esto, primero se **construye** un índice asociativo en cada  $H_{s_i}$  y luego se **prueba** (es decir, se busca en  $H_{s_i}$ ) con las tuplas de  $H_{r_i}$ . La relación  $s$  es la **entrada de construcción** y  $r$  es la **entrada de prueba**.

El índice asociativo en  $H_{s_i}$  se crea en la memoria, así que no es necesario acceder al disco para recuperar las tuplas. La función de asociación utilizada para construir este índice asociativo debe ser distinta de la función de asociación  $h$  utilizada anteriormente, pero aún se aplica exclusivamente a los atributos de la reunión. A lo largo de la reunión en bucle anidado indexada, el sistema utiliza este índice asociativo para recuperar los registros que concuerden con los registro de la entrada de prueba.

Las etapas de construcción y prueba necesitan solamente un único ciclo a través de las entradas de construcción y prueba. La extensión del algoritmo de reunión por asociación para calcular equirreuniones más generales es directa.

Se tiene que elegir el valor de  $n_h$  lo bastante grande como para que, para cada  $i$ , las tuplas de la partición  $H_{s_i}$  de la relación de construcción, junto con el índice asociativo de la partición, quepan en memoria. Pero no tienen por qué caber en memoria las particiones de la relación que se prueban. Sería obviamente mejor utilizar la relación de entrada más pequeña como la relación de construcción. Si el tamaño de la relación de construcción es de  $b_s$  bloques, entonces para que cada una de las  $n_h$  particiones tengan un tamaño menor o igual que  $M$ ,  $n_h$  debe ser al menos  $\lceil b_s/M \rceil$ . Con más exactitud, hay que tener en cuenta también el espacio adicional ocupado por el índice asociativo de la partición, incrementando

```

/* División de s */
for each tupla t_s in s do begin
 $i := h(t_s[AtribsReunión]);$
 $H_{s_i} := H_{s_i} \cup \{t_s\};$
end
/* División de r */
for each tupla t_r in r do begin
 $i := h(t_r[AtribsReunión]);$
 $H_{r_i} := H_{r_i} \cup \{t_r\};$
end
/* Realizar la reunión de cada partición */
for $i := 0$ to n_h do begin
 leer H_{s_i} y construir un índice asociativo en memoria en él
 for each tupla t_r in H_{r_i} do begin
 explorar el índice asociativo en H_{s_i} para localizar todas las tuplas t_s
 tales que $t_s[AtribsReunión] = t_r[AtribsReunión]$
 for each tupla t_s que concuerde in H_{s_i} do begin
 añadir $t_r \bowtie t_s$ al resultado
 end
 end
end
end

```

**Figura 13.9** Reunión por asociación.

$n_h$  según corresponda. Por simplificar, en estos análisis se ignoran muchas veces los requisitos de espacio del índice asociativo.

### 13.5.5.2 División recursiva

Si el valor de  $n_h$  es mayor o igual que el número de marcos de página de la memoria, la división de las relaciones no se puede hacer en un solo ciclo, puesto que no habría suficientes páginas para memorias intermedias. En lugar de eso, la división se tiene que hacer mediante la repetición de ciclos. En un ciclo se puede dividir la entrada en tantas particiones como marcos de página haya disponibles para utilizarlos como memorias intermedias de salida. Cada cajón generado por un ciclo se lee de manera separada y se divide de nuevo en el siguiente ciclo para crear particiones más pequeñas. La función de asociación utilizada en un ciclo es, por supuesto, diferente de la que se ha utilizado en el ciclo anterior. Se repite esta división de la entrada hasta que cada partición de la entrada de construcción quepa en memoria. Esta división se denomina **división recursiva**.

Una relación no necesita de la división recursiva si  $M > n_h + 1$  o, lo que es equivalente,  $M > (b_s/M) + 1$ , lo cual se simplifica (de manera aproximada) a  $M > \sqrt{b_s}$ . Por ejemplo, considerando un tamaño de la memoria de 12 megabytes, dividido en bloques de 4 megabytes, la relación contendría un total de 3K (3.072) bloques. Se puede utilizar una memoria de este tamaño para dividir relaciones de 3K \* 3K bloques, que son 36 gigabytes. Del mismo modo, una relación del tamaño de 1 gigabyte necesita  $\sqrt{256K}$  bloques, o 2 megabytes, para evitar la división recursiva.

### 13.5.5.3 Gestión de desbordamientos

Se produce el **desbordamiento de una tabla de asociación** en la partición  $i$  de la relación de construcción  $s$ , si el índice asociativo de  $H_{s_i}$  es mayor que la memoria principal. El desbordamiento de la tabla de asociación puede ocurrir si existen muchas tuplas en la relación de construcción con los mismos valores en los atributos de la reunión, o si la función de asociación no tiene las propiedades de aleatoriedad y uniformidad. En cualquier caso, algunas de las particiones tendrán más tuplas que la media, mientras que otras tendrán menos; se dice entonces que la división está **sesgada**.

Se puede controlar parcialmente el sesgo mediante el incremento del número de particiones, de tal manera que el tamaño esperado de cada partición (incluido el índice asociativo en la partición) sea algo

menor que el tamaño de la memoria. Por consiguiente, el número de particiones se incrementa en un pequeño valor llamado **factor de escape**, que normalmente es del orden del 20 por ciento del número de particiones calculadas, como se describe en el Apartado 13.5.5.

Aunque se sea conservador con los tamaños de las particiones utilizando un factor de escape, todavía pueden ocurrir desbordamientos. Los desbordamientos de la tabla de asociación se pueden tratar mediante *resolución del desbordamiento* o *evitación del desbordamiento*. La **resolución del desbordamiento** se realiza como sigue. Si  $H_{s_i}$ , para cualquier  $i$ , resulta ser demasiado grande, se divide de nuevo en particiones más pequeñas utilizando una función de asociación distinta. Del mismo modo, también se divide  $H_{r_i}$  utilizando la nueva función de asociación, y solamente es necesario reunir las tuplas en las particiones concordantes.

En cambio, la **evitación del desbordamiento** realiza la división más cuidadosamente, de tal manera que el desbordamiento nunca se produce en la fase de construcción. Para evitar el desbordamiento se implementa la división de la relación de construcción  $s$  en muchas particiones pequeñas inicialmente, para luego combinar algunas de estas particiones de tal manera que cada partición combinada quepa en memoria. Además, la relación de prueba  $r$  se tiene que combinar de la misma manera que se combinan las particiones de  $s$ , sin importar los tamaños de  $H_{r_i}$ .

Las técnicas de resolución y evitación podrían fallar en algunas particiones, si un gran número de las tuplas en  $s$  tuvieran el mismo valor en los atributos de la reunión. En ese caso, en lugar de crear un índice asociativo en memoria y utilizar una reunión en bucle anidado para reunir las particiones, se pueden utilizar otras técnicas de reunión, tales como la reunión en bucle anidado por bloques, en esas particiones.

### 13.5.4 Coste de la reunión por asociación

Se considera ahora el coste de una reunión por asociación. El análisis supone que no hay desbordamiento de la tabla de asociación. Primero se considera el caso en el que no es necesaria una división recursiva.

- La división de las dos relaciones  $r$  y  $s$  reclama una lectura completa de ambas así como su posterior escritura. Esta operación necesita  $2(b_r + b_s)$  transferencias de bloques, donde  $b_r$  y  $b_s$  denotan respectivamente el número de bloques que contienen registros de las relaciones  $r$  y  $s$ . Las fases de construcción y prueba leen cada una de las particiones una vez, empleando  $b_r + b_s$  transferencias adicionales. El número de bloques ocupados por las particiones podría ser ligeramente mayor que  $b_r + b_s$  debido a que los bloques están parcialmente ocupados. El acceso a estos bloques llenos en parte puede añadir un gasto adicional de  $2n_h$  a lo sumo, ya que cada una de las  $n_h$  particiones podría tener un bloque lleno parcialmente que haya que escribir y leer de nuevo. Así, el coste estimado para una reunión por asociación es

$$3(b_r + b_s) + 4n_h$$

transferencias de bloques. La sobrecarga  $4n_h$  es muy pequeña comparada con  $b_r + b_s$  y se puede ignorar.

- Si se da por supuesto que se asignan  $b_b$  bloques para las memorias intermedias de entrada y salida, la división necesita  $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$  búsquedas. Las fases de construcción y prueba requieren sólo una búsqueda para cada una de las  $n_h$  particiones de cada relación, ya que cada partición se puede leer secuencialmente. Por tanto, la reunión por asociación necesita  $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$  búsquedas.

Considérese ahora el caso en el que es necesario realizar la división recursiva. Cada ciclo reduce el tamaño de cada una de las particiones por un factor esperado de  $M - 1$ ; y los ciclos se repiten hasta que cada partición tenga como mucho un tamaño de  $M$  bloques. Por tanto, el número esperado de ciclos necesarios para dividir  $s$  es  $\lceil \log_{M-1}(b_s) - 1 \rceil$ .

- Como todos los bloques de  $s$  se leen y se escriben en cada ciclo, el número total de transferencias de bloques para dividir  $s$  es  $2b_s \lceil \log_{M-1}(b_s) - 1 \rceil$ . Como el número de ciclos para dividir  $r$  es el

mismo que el número de ciclos para dividir  $s$ , por tanto la estimación del coste de la reunión es

$$2(b_r + b_s)[\log_{M-1}(b_s) - 1] + b_r + b_s$$

transferencias de bloques.

- De nuevo, dando por supuesto que se asignen  $b_b$  bloques en memoria intermedia para cada partición, e ignorando el número relativamente pequeño de búsquedas durante las fases de construcción y prueba, la reunión por asociación con división recursiva necesita

$$2([\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil])[\log_{M-1}(b_s) - 1]$$

búsquedas.

Considérese, por ejemplo, la reunión  $cliente \bowtie impositor$ . Con un tamaño de memoria de 20 bloques, se puede dividir *impositor* en cinco partes, cada una de 20 bloques, con un tamaño tal que caben en memoria. Así, sólo es necesario un ciclo para la división. De la misma manera la relación *cliente* se divide en cinco particiones, cada una de 80 bloques. Si se ignora el coste de escribir los bloques parcialmente llenos, el coste es  $3(100 + 400) = 1.500$  transferencias de bloques. Hay suficiente memoria para asignar tres bloques a la entrada y cada uno de los cinco bloques de salida durante la división, lo que resulta en  $2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$  búsquedas.

Es posible mejorar la reunión por asociación si el tamaño de la memoria principal es grande. Cuando la entrada de construcción se puede guardar por completo en memoria principal hay que establecer  $n_h$  a 0; de este modo el algoritmo de reunión por asociación se ejecuta rápidamente, sin dividir las relaciones en archivos temporales y sin importar el tamaño de la entrada de prueba. El coste estimado desciende a  $b_r + b_s$  transferencias de bloques y dos búsquedas.

### 13.5.5.5 Reunión por asociación híbrida

El algoritmo de **reunión por asociación híbrida** realiza otra optimización; es útil cuando los tamaños de la memoria son relativamente grandes pero no cabe toda la relación de construcción en memoria. La fase de división del algoritmo de reunión por asociación necesita un bloque de memoria como memoria intermedia para cada partición que se cree, más un bloque de memoria como memoria intermedia de entrada. Por tanto, se necesitan  $n_h + 1$  bloques de memoria para dividir las dos relaciones. Si la memoria es mayor que  $n_h + 1$ , se puede utilizar el resto de la memoria ( $M - n_h - 1$  bloques) para guardar en memorias intermedias la primera partición de la entrada de construcción (esto es,  $H_{s_0}$ ), así que no es necesario escribirla ni leerla de nuevo. Más aún, la función de asociación se diseña de tal manera que el índice asociativo en  $H_{s_0}$  quepa en  $M - n_h - 1$  bloques, así que, al final de la división de  $s$ ,  $H_{s_0}$  está en memoria por completo y se puede construir un índice asociativo en  $H_{s_0}$ .

Cuando  $r$  se divide de nuevo, las tuplas en  $H_{r_0}$  no se escriben en disco; en su lugar, según se van generando, el sistema las utiliza para examinar el índice asociativo residente en memoria de  $H_{s_0}$  y para generar las tuplas de salida de la reunión. Después de utilizarlas para la prueba, se descartan las tuplas, así que la partición  $H_{r_0}$  no ocupa ningún espacio en memoria. De este modo se ahorra un acceso de lectura y otro de escritura para cada bloque de  $H_{r_0}$  y  $H_{s_0}$ . Las tuplas en otras particiones se escriben de la manera usual para reunirlas más tarde. El ahorro de la reunión por asociación híbrida puede ser importante si la entrada de construcción es ligeramente mayor que la memoria.

Si el tamaño de la relación de construcción es  $b_s$ ,  $n_h$  es aproximadamente igual a  $b_s/M$ . Así, la reunión por asociación híbrida es más útil si  $M >> b_s/M$  o si  $M >> \sqrt{b_s}$ , donde la notación  $>>$  significa *mucho mayor que*. Por ejemplo, supóngase que el tamaño del bloque es de 4 kilobytes y que el tamaño de la relación de construcción es de 1 gigabyte. Entonces, el algoritmo híbrido de reunión por asociación es útil si el tamaño de la memoria es claramente mayor que 2 megabytes; las memorias de 100 megabytes o más son comunes en las computadoras de hoy en día.

Considérese de nuevo la reunión  $cliente \bowtie impositor$ . Con una memoria de 25 bloques de tamaño, se puede dividir *impositor* en cinco particiones, cada una de 20 bloques, y con la primera de las particiones de la relación de construcción almacenada en memoria. Ocupa 20 bloques de memoria; un bloque se utiliza para la entrada y cuatro bloques más para guardar en memorias intermedias cada partición. De manera similar se divide la relación *cliente* en cinco particiones cada una de tamaño 80, la primera de

las cuales de utiliza precisamente para comprobar, en lugar de escribirse y leerse de nuevo. Ignorando el coste de escribir los bloques parcialmente llenos, el coste es de  $3(80 + 320) + 20 + 80 = 1.300$  bloques transferidos, en lugar de las 1.500 transferencias de bloques sin la optimización por asociación híbrida. Sin embargo, en este caso el tamaño de la memoria intermedia para la entrada y para cada partición escrita a disco desciende a un bloque, aumentando el número de búsquedas a alrededor del número de accesos a bloques, y aumentando también el coste total. Por tanto, en este caso es mejor usar una reunión por asociación con una memoria intermedia mayor de  $b_b$  en lugar de usar la reunión por asociación híbrida. Sin embargo, con tamaños mucho mayores de memoria, el aumento de  $b_b$  más allá de un punto determinado disminuye el rendimiento, y la memoria restante se puede usar para implementar la reunión por asociación híbrida.

### 13.5.6 Reuniones complejas

Las reuniones en bucle anidado y en bucle anidado por bloques son útiles sean cuales sean las condiciones de la reunión. Las otras técnicas de reunión son más eficientes que las reuniones en bucle anidado y sus variantes, aunque sólo se pueden utilizar con condiciones simples, tales como las reuniones naturales o las equirreuniones. Se pueden implementar reuniones con condiciones de la reunión más complejas, tales como conjunciones y disyunciones, utilizando las técnicas eficientes de la reunión mediante la aplicación de las técnicas desarrolladas en el Apartado 13.3.4 para el manejo de selecciones complejas.

Considérese la siguiente reunión con una condición conjuntiva:

$$r \bowtie_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n} s$$

Se pueden aplicar una o más de las técnicas de reunión descritas anteriormente en cada condición individual  $r \bowtie_{\theta_1} s$ ,  $r \bowtie_{\theta_2} s$ ,  $r \bowtie_{\theta_3} s$ , y así sucesivamente. El resultado total de la reunión se determina calculando primero el resultado de una de estas reuniones simples  $r \bowtie_{\theta_i} s$ ; cada par de tuplas del resultado intermedio se compone de una tupla de  $r$  y otra de  $s$ . El resultado total de la reunión consiste en las tuplas del resultado intermedio que satisfacen el resto de condiciones

$$\theta_1 \wedge \dots \wedge \theta_{i-1} \wedge \theta_{i+1} \wedge \dots \wedge \theta_n$$

Estas condiciones se pueden ir comprobando según se generan las tuplas de  $r \bowtie_{\theta_i} s$ .

Una reunión cuya condición es una disyunción se puede calcular como se indica a continuación. Considérese

$$r \bowtie_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n} s$$

La reunión se puede calcular como la unión de los registros de las reuniones  $r \bowtie_{\theta_i} s$ :

$$(r \bowtie_{\theta_1} s) \cup (r \bowtie_{\theta_2} s) \cup \dots \cup (r \bowtie_{\theta_n} s)$$

Los algoritmos para calcular la unión de relaciones se describen en el Apartado 13.6.

## 13.6 Otras operaciones

Otras operaciones relacionales y operaciones relacionales extendidas (como eliminación de duplicados, proyección, operaciones sobre conjuntos, reunión externa y agregación) se pueden implementar según se describe en los Apartados 13.6.1 al 13.6.5.

### 13.6.1 Eliminación de duplicados

Se puede implementar fácilmente la eliminación de duplicados utilizando la ordenación. Las tuplas idénticas aparecerán consecutivas durante la ordenación, pudiéndose eliminar todas las copias menos una. Con la ordenación-mezcla externa, se pueden eliminar los duplicados mientras se crea una secuencia antes de que ésta se escriba en el disco, reduciendo así el número de bloques transferidos. El resto de duplicados se pueden suprimir durante la etapa de reunión/mezcla, así que el resultado final está libre de repeticiones. El coste estimado en el peor de los casos para la eliminación de duplicados es el mismo que el coste estimado en el peor caso para la ordenación de una relación.

Se puede implementar también la eliminación de duplicados utilizando la asociación de una manera similar al algoritmo de reunión por asociación. Primero, se divide la relación basándose en una función

de asociación en la tupla entera. Luego, se lee cada partición, y se construye un índice asociativo en memoria. Mientras se construye el índice asociativo se inserta una tupla solamente si no estaba ya presente. En otro caso se descarta. Después de que todas las tuplas de la relación se hayan procesado, las tuplas en el índice asociativo se escriben al resultado. El coste estimado es el mismo que el coste de procesar (división y lectura de cada partición) la relación de construcción en una reunión por asociación.

Debido al coste relativamente alto de la eliminación de duplicados, SQL exige una petición explícita del usuario para suprimir los duplicados; en caso contrario, se conservan.

### 13.6.2 Proyección

La proyección se puede implementar fácilmente realizando la proyección de cada tupla, pudiendo originar una relación con registros repetidos y suprimiendo después los registros duplicados. La eliminación de duplicados se puede llevar a cabo según se ha descrito en el Apartado 13.6.1. Si los atributos de la lista de proyección incluyen una clave de la relación, no se producirán duplicados; por tanto no será necesario eliminarlos. La proyección generalizada (tratada en el Apartado 2.4.1) se puede implementar de la misma manera que las proyecciones.

### 13.6.3 Operaciones sobre conjuntos

Las operaciones *unión*, *intersección* y *diferencia de conjuntos* se pueden implementar ordenando primero ambas relaciones y examinando después cada relación para producir el resultado. En  $r \cup s$ , cuando una exploración concurrente en ambas relaciones descubre la misma tupla en los dos archivos, solamente se conserva una de las tuplas. Por otra parte, el resultado de  $r \cap s$  contendrá únicamente aquellas tuplas que aparezcan en ambas relaciones. De la misma manera se implementa la *diferencia de conjuntos*,  $r - s$ , guardando aquellas tuplas de  $r$  que estén ausentes de  $s$ .

Para todas estas operaciones solamente se necesita una exploración en cada relación de entrada, así el coste es  $b_r + b_s$  transferencias de bloques si las relaciones están ordenadas de igual forma. Suponiendo que en el peor caso sólo hay un bloque de memoria intermedia para cada relación, se necesitarían un total de  $b_r + b_s$  búsquedas además de las  $b_r + b_s$  transferencias de bloques. El número de búsquedas se puede reducir asignando bloques adicionales de memoria intermedia.

Si las relaciones no están ordenadas inicialmente, hay que incluir el coste de la ordenación. Se puede utilizar cualquier otro orden en la evaluación de la operación de conjuntos, siempre que las dos entradas tengan la misma ordenación.

La asociación proporciona otra manera de implementar estas operaciones sobre conjuntos. El primer paso en cada caso es dividir las dos relaciones utilizando la misma función de asociación y de este modo crear las particiones  $H_{r_0}, H_{r_1}, \dots, H_{r_{n_h}}$  y  $H_{s_0}, H_{s_1}, \dots, H_{s_{n_h}}$ . En función de cada operación, el sistema da a continuación estos pasos en cada partición  $i = 0, 1, \dots, n_h$ :

- $r \cup s$ 
  1. Construir un índice asociativo en memoria sobre  $H_{r_i}$ .
  2. Añadir las tuplas de  $H_{s_i}$  al índice asociativo solamente si no estaban ya presentes.
  3. Añadir las tuplas del índice asociativo al resultado.
- $r \cap s$ 
  1. Construir un índice asociativo en memoria en  $H_{r_i}$ .
  2. Para cada tupla en  $H_{s_i}$  probar el índice asociativo y pasar la tupla al resultado únicamente si ya estaba presente en el índice.
- $r - s$ 
  1. Construir un índice asociativo en memoria en  $H_{r_i}$ .
  2. Para cada tupla de  $H_{s_i}$  probar el índice asociativo y, si la tupla está presente en el índice, suprimirla del índice asociativo.
  3. Añadir las tuplas restantes del índice asociativo al resultado.

### 13.6.4 Reunión externa

Recordemos las *operaciones de reunión externa* que se describieron en el Apartado 2.4.3. Por ejemplo, la reunión externa por la izquierda  $cliente \bowtie_{\theta} impositor$  contiene la reunión de *cliente* y de *impositor* y además, para cada tupla  $t$  de *cliente* que no concuerde con alguna tupla en *impositor* (es decir, donde no esté *nombre\_cliente* en *impositor*), se añade la siguiente tupla  $t_1$  al resultado. Para todos los atributos del esquema de *cliente* la tupla  $t_1$  tiene los mismos valores que la tupla  $t$ . El resto de los atributos (del esquema de *impositor*) de la tupla  $t_1$  contienen el valor nulo.

Se pueden implementar las operaciones de reunión externa empleando una de las dos estrategias siguientes:

1. Calcular la reunión correspondiente, y luego añadir más tuplas al resultado de la reunión hasta obtener la reunión externa resultado. Considérese la operación de reunión externa por la izquierda y dos relaciones:  $r(R)$  y  $s(S)$ . Para evaluar  $r \bowtie_{\theta} s$ , se calcula primero  $r \bowtie_{\theta} s$  y se guarda este resultado como relación temporal  $q_1$ . A continuación se calcula  $r - \Pi_R(q_1)$ , que produce las tuplas de  $r$  que no participaron en la reunión. Se puede utilizar cualquier algoritmo para calcular las reuniones. Luego se llenan cada una de estas tuplas con valores nulos en los atributos de  $s$  y se añaden a  $q_1$  para obtener el resultado de la reunión externa.

La operación reunión externa por la derecha  $r \bowtie_{\theta} s$  es equivalente a  $s \bowtie_{\theta} r$  y, por tanto, se puede implementar de manera simétrica a la reunión externa por la izquierda. También se puede implementar la operación reunión externa completa  $r \bowtie_{\theta} s$  calculando la reunión  $r \bowtie s$  y añadiendo luego las tuplas adicionales de las operaciones reunión externa por la izquierda y por la derecha, al igual que antes.

2. Modificar los algoritmos de la reunión. Así, es fácil extender los algoritmos de reunión en bucle anidado para calcular la reunión externa por la izquierda. Las tuplas de la relación externa que no concuerdan con ninguna tupla de la relación interna se escriben en la salida después de haber sido completadas con valores nulos. Sin embargo, es difícil extender la reunión en bucle anidado para calcular la reunión externa completa.

Las reuniones externas naturales y las reuniones externas con una condición de equirreunión se pueden calcular mediante extensiones de los algoritmos de reunión por mezcla y reunión por asociación. La reunión por mezcla se puede extender para realizar la reunión externa completa como se muestra a continuación. Cuando se está produciendo la mezcla de las dos relaciones, las tuplas de la relación que no encajan con ninguna tupla de la otra relación se pueden completar con valores nulos y escribirse en la salida. Del mismo modo se puede extender la reunión por mezcla para calcular las reuniones externas por la izquierda y por la derecha mediante la copia de las tuplas que no concuerden (rellenadas con valores nulos) desde solamente una de las relaciones. Puesto que las relaciones están ordenadas, es fácil detectar si una tupla coincide o no con alguna de las tuplas de la otra relación. Por ejemplo, cuando se hace una reunión por mezcla de *cliente* e *impositor*, las tuplas se leen según el orden de *nombre\_cliente* y es fácil comprobar para cada tupla si hay alguna tupla coincidente.

El coste estimado para implementar reuniones externas utilizando el algoritmo de reunión por mezcla es el mismo que para la correspondiente reunión. La única diferencia está en el tamaño del resultado y, por tanto, en los bloques transferidos para copiarlos que no se han tenido en cuenta en las estimaciones anteriores del coste.

La extensión del algoritmo de reunión por asociación para calcular reuniones externas se deja como ejercicio (Ejercicio 13.13).

### 13.6.5 Agregación

Considérese el operador de agregación  $\mathcal{G}$  estudiado en el Apartado 2.4.2. Por ejemplo, la operación:

$$nombre\_sucursal \mathcal{G}_{sum(saldo)}(cuenta)$$

agrupa tuplas de *cuenta* por *sucursal* y calcula el saldo total de todas las cuentas de cada sucursal.

La operación agregación se puede implementar de una manera parecida a la eliminación de duplicados. Se utiliza la ordenación o la asociación, al igual que se hizo para la supresión de duplicados, pero

basándose ahora en la agrupación de atributos (*nombre\_sucursal* en el ejemplo anterior). Sin embargo, en lugar de eliminar las tuplas con el mismo valor en los atributos de la agrupación, se reúnen en grupos y se aplican las operaciones agregación en cada grupo para obtener el resultado.

El coste estimado para la implementación de la operación agregación es el mismo coste de la eliminación de duplicados para las funciones de agregación como **min**, **max**, **sum**, **count** y **avg**.

En lugar de reunir todas las tuplas en grupos y aplicar entonces las funciones de agregación, se pueden implementar las funciones de agregación **sum**, **min**, **max**, **count** y **avg** sobre la marcha según se construyen los grupos. Para el caso de **sum**, **min** y **max**, cuando se encuentran dos tuplas del mismo grupo el sistema las sustituye por una sola tupla que contenga **sum**, **min** o **max**, respectivamente, de las columnas que se están agregando. Para la operación **count**, se mantiene una cuenta incremental para cada grupo con tuplas descubiertas. Por último, la operación **avg** se implementa calculando la suma y la cuenta de valores sobre la marcha, para dividir finalmente la suma entre la cuenta para obtener la media.

Si todas las tuplas del resultado caben en memoria, las implementaciones basadas en ordenación y las basadas en asociación no necesitan escribir ninguna tupla en disco. Según se leen las tuplas se pueden insertar en un estructura ordenada de árbol o en un índice asociativo. Así, cuando se utilizan las técnicas de agregación sobre la marcha, solamente es necesario almacenar una tupla para cada uno de los grupos. Por tanto, la estructura ordenada de árbol o el índice asociativo caben en memoria y se puede procesar la agregación con sólo  $b_r$  transferencias de bloques (y una búsqueda), en lugar de las  $3b_r$  transferencias (y en el peor caso de hasta  $2b_r$  búsquedas) que se necesitarían en el otro caso.

## 13.7 Evaluación de expresiones

Hasta aquí se ha estudiado cómo llevar a cabo operaciones relacionales individuales. Ahora se considera cómo evaluar una expresión que contiene varias operaciones. La manera evidente de evaluar una expresión es simplemente evaluar una operación a la vez en un orden apropiado. El resultado de cada evaluación se **materializa** en una relación temporal para su inmediata utilización. Un inconveniente de esta aproximación es la necesidad de construir relaciones temporales, que (a menos que sean pequeñas) se tienen que escribir en disco. Un enfoque alternativo es evaluar varias operaciones de manera simultánea en un **cauce**, con los resultados de una operación pasados a la siguiente, sin la necesidad de almacenar relaciones temporales.

En los Apartados 13.7.1 y 13.7.2 se consideran ambos enfoques, *materialización* y *encauzamiento*. Se verá que el coste de estos enfoques puede diferir substancialmente, pero también que existen casos donde sólo la materialización es posible.

### 13.7.1 Materialización

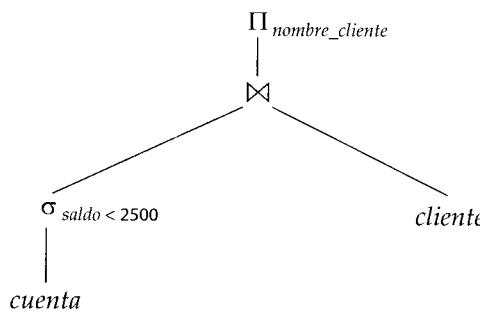
Intuitivamente es fácil de entender cómo evaluar una expresión observando una representación gráfica de la expresión en un **árbol de operadores**. Considérese la expresión

$$\Pi_{\text{nombre\_cliente}} (\sigma_{\text{saldo} < 2500} (\text{cuenta}) \bowtie \text{cliente})$$

que se muestra en la Figura 13.10.

Si se aplica el enfoque de la materialización, se comienza por las operaciones de la expresión de nivel más bajo (al fondo del árbol). En el ejemplo solamente hay una de estas operaciones: la operación selección en *cuenta*. Las entradas de las operaciones de nivel más bajo son las relaciones de la base de datos. Se ejecutan estas operaciones utilizando los algoritmos ya estudiados, almacenando sus resultados en relaciones temporales. Luego se utilizan estas relaciones temporales para ejecutar las operaciones del siguiente nivel en el árbol, cuyas entradas son ahora o bien relaciones temporales o bien relaciones almacenadas en la base de datos. En este ejemplo, las entradas de la reunión son la relación *cliente* y la relación temporal producida por la selección en *cuenta*. Ahora se puede evaluar la reunión creando otra relación temporal.

Repetiendo este proceso se calcularía finalmente la operación en la raíz del árbol, obteniendo el resultado final de la expresión. En el ejemplo se consigue el resultado final mediante la ejecución de la operación proyección de la raíz utilizando como entrada la relación temporal creada por la reunión.



**Figura 13.10** Representación gráfica de una expresión.

Una evaluación como la descrita se llama **evaluación materializada**, puesto que los resultados de cada operación intermedia se crean (materializan) para utilizarse a continuación en la evaluación de las operaciones del siguiente nivel.

El coste de una evaluación materializada no es simplemente la suma de los costes de las operaciones involucradas. Cuando se calcularon los costes estimados de los algoritmos se ignoró el coste de escribir el resultado de la operación en disco. Para calcular el coste de evaluar una expresión como la que se ha hecho hay que añadir los costes de todas las operaciones, incluyendo el coste de escribir los resultados intermedios en disco. Supóngase que los registros del resultado se acumulan en una memoria intermedia y que cuando ésta se llena, los registros se escriben en el disco. El número de bloques escritos,  $b_r$ , se puede estimar en  $n_r/f_r$ , donde  $n_r$  es el número aproximado de tuplas de la relación resultado  $r$  y  $f_r$  es el factor de bloqueo de la relación resultado, es decir, el número de registros de  $r$  que caben en un bloque. Además del tiempo de transferencia es posible que se necesiten algunas búsquedas, ya que la cabeza del disco se puede haber desplazado entre escrituras sucesivas. Se puede estimar el número de búsquedas como  $\lceil b_r/b_b \rceil$ , donde  $b_b$  es el tamaño en bloques de la memoria intermedia para la salida.

La **memoria intermedia doble** (usando dos memorias intermedias, una donde progresiva la ejecución del algoritmo mientras que la otra se está copiando) permite que el algoritmo se ejecute más rápidamente mediante la ejecución en paralelo de acciones en la CPU con acciones de E/S. El número de búsquedas se puede reducir asignando bloques adicionales en la memoria intermedia de salida y escribiendo varios bloques a la vez.

## 13.7.2 Encauzamiento

Se puede mejorar la eficiencia de la evaluación de consultas mediante la reducción del número de archivos temporales que se producen. Se lleva a cabo esta reducción con la combinación de varias operaciones relacionales en un *encauzamiento* de operaciones, en el que se pasan los resultados de una operación a la siguiente operación del encauzamiento. Esta evaluación, como se ha descrito, se denomina **evaluación encauzada**. La combinación de operaciones en un encauzamiento elimina el coste de leer y escribir relaciones temporales.

Por ejemplo, considérese la reunión de un par de relaciones seguida de una proyección ( $\Pi_{a1,a2}(r \bowtie s)$ ). Si se aplicara la materialización, la evaluación implicaría la creación de una relación temporal para guardar el resultado de la reunión y la posterior lectura del resultado para realizar la proyección. Estas operaciones se pueden combinar como sigue. Cuando la operación reunión genera una tupla del resultado, se pasa inmediatamente esa tupla al operador de proyección para su procesamiento. Mediante la combinación de la reunión y de la proyección, se evita la creación de resultados intermedios, creando en su lugar el resultado final directamente.

### 13.7.2.1 Implementación del encauzamiento

Se puede implementar el encauzamiento construyendo una única y compleja operación que combine las operaciones que constituyen el encauzamiento. Aunque este enfoque podría ser factible en muchas situaciones, es deseable en general reusar el código en operaciones individuales en la construcción del encauzamiento. Por lo tanto, cada operación del encauzamiento se modela como un proceso aislado o

una hebra en el sistema, que toma un flujo de tuplas de sus entradas encauzadas y produce un flujo de tuplas como salida. Para cada pareja de operaciones adyacentes en el encauzamiento se crea una memoria intermedia para guardar las tuplas que se envían de una operación a la siguiente.

En el ejemplo de la Figura 13.10, las tres operaciones se pueden situar en un encauzamiento, en el que los resultados de la selección se pasan a la reunión según se generan. Por su parte, los resultados de la reunión se envían a la proyección según se van generando. Así, los requisitos de memoria son bajos, ya que los resultados de una operación no se almacenan por mucho tiempo. Sin embargo, como resultado del encauzamiento, las entradas de las operaciones no están disponibles todas a la vez para su procesamiento.

Los encauzamientos se pueden ejecutar de alguno de los siguientes modos:

1. Bajo demanda.
2. Desde los productores.

En un **encauzamiento bajo demanda**, el sistema reitera peticiones de tuplas desde la operación de la cima del encauzamiento. Cada vez que una operación recibe una petición de tuplas, calcula la siguiente tupla (o tuplas) a devolver y la envía. Si las entradas de la operación no están encauzadas, la(s) siguiente(s) tupla(s) a devolver se calcula(n) de las relaciones de entrada mientras se lleva cuenta de lo que se ha remitido hasta el momento. Si alguna de sus entradas está encauzada, la operación también hace peticiones de tuplas desde sus entradas encauzadas. Así, utilizando las tuplas recibidas en sus entradas encauzadas, la operación calcula sus tuplas de salida y las envía hasta su padre.

En un **encauzamiento por los productores**, los operadores no esperan a que se produzcan peticiones para producir tuplas, en su lugar generan las tuplas **impacientemente**. Cada operación del fondo del encauzamiento genera continuamente tuplas de salida y las ponen en su memoria intermedia de salida hasta que se llena. Una operación en cualquier otro nivel del encauzamiento obtiene sus tuplas de entrada de un nivel inferior del encauzamiento hasta llenar su memoria intermedia de salida. Una vez que la operación ha utilizado una tupla de su entrada encauzada, la elimina de ella. En cualquier caso, una vez que la memoria intermedia de salida esté llena, la operación esperará hasta que su operación padre elimine las tuplas de la memoria intermedia para hacer más espacio a nuevas tuplas. En este momento, la operación genera más tuplas hasta que se llene la memoria intermedia de nuevo. Este proceso se repite por una operación hasta que se hayan generado todas las tuplas de salida.

El sistema necesita cambiar de una operación a otra solamente cuando se llena una memoria intermedia de salida o cuando una memoria intermedia de entrada está vacía y se necesitan más tuplas para generar las tuplas de salida. En un sistema de procesamiento paralelo, las operaciones del encauzamiento se pueden ejecutar concurrentemente en distintos procesadores (véase el Capítulo 21).

Se puede imaginar el encauzamiento por los productores como una **inserción** de datos de abajo hacia arriba en el árbol de operaciones, mientras que el encauzamiento bajo demanda se puede pensar como una **extracción** de datos desde la cima del árbol de operaciones. Mientras que las tuplas se generan *impacientemente* en el encauzamiento por los productores, se generan de forma **perezosa**, bajo demanda, en el encauzamiento bajo demanda.

Cada operación en un encauzamiento bajo demanda se puede implementar como un **iterador**, que proporciona las siguientes funciones: *abrir()*, *siguiente()* y *cerrar()*. Después de una llamada a *abrir()*, cada llamada a *siguiente()* devuelve la siguiente tupla de salida de la operación. La implementación de la operación realiza por turno llamadas a *abrir()* y a *siguiente()* desde sus entradas, cuando necesita tuplas de entrada. La función *cerrar()* comunica al iterador que no necesita más tuplas. De este modo, el iterador mantiene el **estado** de su ejecución entre las llamadas, de tal manera que las sucesivas peticiones *siguiente()* reciban sucesivas tuplas resultado.

Por ejemplo, en un iterador que implemente la operación selección usando la búsqueda lineal, la operación *abrir()* inicia una exploración de archivo y el estado del iterador registra el punto en el que el archivo se ha explorado. Cuando se llama a la función *siguiente()* la exploración del archivo continúa a continuación del punto anterior; cuando se encuentre la siguiente tupla que cumpla la selección al explorar el archivo, se devuelve la tupla después de almacenar el punto donde se encontró en el estado del iterador. Una operación *abrir()* del iterador de reunión por mezcla abriría sus entradas y, si aún no están ordenadas, también las ordenaría. En las llamadas a *siguiente()* se devolvería el siguiente par

de tuplas coincidentes. La información de estado consistiría en el grado en que se ha explorado cada entrada.

Los detalles de la implementación de iteradores se dejan para completar en el Ejercicio práctico 13.7. El encauzamiento bajo demanda se utiliza normalmente más que el encauzamiento por los productores, ya que es más fácil de implementar.

### 13.7.2.2 Algoritmos de evaluación para el encauzamiento

Considérese una operación reunión cuya entrada del lado izquierdo está encauzada. Puesto que está encauzada, no toda la entrada está disponible al mismo tiempo para el procesamiento de la operación reunión. Al no estar disponible, se limita la elección del algoritmo de reunión a emplear. Por ejemplo, no se puede usar la reunión por mezcla si las entradas no están ordenadas, puesto que no es posible ordenar la relación hasta que todas las tuplas estén disponibles (así, en efecto, se convierte el encauzamiento en materialización). Sin embargo, se puede utilizar la reunión en bucle anidado indexada: según se reciben las tuplas por el lado izquierdo de la reunión se pueden utilizar para indexar el lado derecho de la relación y para generar las tuplas resultado de la reunión. Este ejemplo ilustra que las elecciones respecto del algoritmo a utilizar para una operación y las elecciones respecto del encauzamiento no son independientes.

Las restricciones en los algoritmos de evaluación que se pueden utilizar es un factor determinante del encauzamiento. Como resultado, a pesar de las aparentes ventajas del encauzamiento, hay casos donde la materialización alcanza un coste total menor. Supóngase que se desea realizar la reunión de  $r$  y de  $s$ , y que la entrada  $r$  está encauzada. Si se utiliza una reunión en bucle anidado indexada para llevar a cabo el encauzamiento, podría ser necesario un acceso a disco para cada tupla de la relación de entrada encauzada. El coste de esta técnica es  $n_r * AA_i * (t_S + t_T)$ , donde  $AA_i$  es la altura del índice de  $s$ . Con la materialización, el coste de copiar  $r$  sería  $b_r * t_T$ . Con una técnica de reunión, como la reunión por asociación, podría ser posible realizar la reunión con un coste aproximado de  $3(b_r + b_s)$  transferencias de bloques más  $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$  búsquedas (asumiendo suficiente memoria para evitar la división recursiva). Para  $n_r$  grandes, la materialización sería más económica.

El uso eficiente del encauzamiento necesita la utilización de algoritmos de evaluación que puedan generar tuplas de salida según se están recibiendo tuplas por las entradas de la operación. Se pueden distinguir dos casos:

1. Solamente una de las entradas de la reunión está encauzada.
2. Las dos entradas de la reunión están encauzadas.

Si únicamente una de las entradas de la reunión está encauzada, la reunión en bucle anidado indexada es la elección más natural. Si las tuplas de entrada encauzadas están ordenadas según los atributos de la reunión y la condición de la reunión es una equirreunión, también se puede emplear la reunión por mezcla. La reunión por asociación híbrida se puede utilizar con la entrada encauzada como la relación de prueba. Sin embargo, las tuplas que no están en la primera partición se enviarán a la salida solamente después de que la relación de entrada encauzada se reciba por completo. La reunión por asociación híbrida es útil si las entradas no encauzadas caben completamente en memoria, o si al menos la mayoría de las entradas caben en memoria.

Si ambas entradas están encauzadas, la elección de los algoritmos de reunión está más limitada. Si ambas entradas están ordenadas en los atributos de la reunión y la condición de la reunión es una equirreunión, entonces se puede usar la reunión por mezcla. Otra técnica alternativa es la **reunión encauzada**, que se muestra en la Figura 13.11. El algoritmo supone que las tuplas de entrada de ambas relaciones,  $r$  y  $s$ , están encauzadas. Las tuplas disponibles de ambas relaciones se dejan listas para su procesamiento en una cola simple. Así mismo, se generan unas entradas de la cola especiales, llamadas  $Fin_r$  y  $Fin_s$ , que sirven como marcas de fin de archivo y que se insertan en la cola después de que se hayan generado todas las tuplas de  $r$  y de  $s$  (respectivamente). Para una evaluación eficaz se deberían construir los índices apropiados en las relaciones  $r$  y  $s$ . Según se añaden tuplas a  $r$  y a  $s$  se deben mantener los índices actualizados.

```

hechor := falso;
hechos := falso;
r := ∅;
s := ∅;
resultado := ∅;
while not hechor or not hechos do
 begin
 if la cola está vacía, then esperar hasta que la cola no esté vacía;
 t := entrada de la cima de la cola;
 if t = Finr then hechor := cierto
 else if t = Fins then hechos := cierto
 else if t es de la entrada r
 then
 begin
 r := r ∪ {t};
 resultado := resultado ∪ ({t} ⊗ s);
 end
 else /* t es de la entrada s */
 begin
 s := s ∪ {t};
 resultado := resultado ∪ (r ⊗ {t});
 end
 end

```

**Figura 13.11** Algoritmo de reunión encauzada.

## 13.8 Resumen

- La primera acción que el sistema debe realizar en una consulta es traducirla en su formato interno, que (para sistemas de bases de datos relacionales) está basado normalmente en el álgebra relacional. En el proceso de generación del formato interno de la consulta, el analizador comproba la sintaxis, verifica que los nombres de relación que figuran en la consulta son nombres de relaciones de la base de datos, etc. Si la consulta se expresó en términos de una vista, el analizador sustituye todas las referencias al nombre de la vista con su expresión del álgebra relacional para calcularla.
- Dada una consulta, generalmente hay diversos métodos para calcular la respuesta. Es responsabilidad del sistema transformar la consulta proporcionada por el usuario en otra consulta equivalente que se pueda ejecutar de un modo más eficiente. El Capítulo 14 estudia la optimización de consultas.
- Se pueden procesar consultas que impliquen selecciones sencillas mediante una búsqueda lineal, una búsqueda binaria o utilizando índices. Así mismo, se pueden manejar selecciones más complejas mediante uniones e intersecciones de los resultados de selecciones simples.
- Se pueden ordenar relaciones que sean más grandes que la memoria utilizando el algoritmo de ordenación-mezcla externa.
- Las consultas que impliquen una reunión natural se pueden procesar de varias maneras, dependiendo de la disponibilidad de índices y del tipo de almacenamiento físico utilizado para las relaciones.
  - Si la reunión resultante es casi tan grande como el producto cartesiano de las dos relaciones, una estrategia de *reunión en bucle anidado por bloques* podría ser ventajosa.
  - Si hay índices disponibles, se puede utilizar la *reunión en bucle anidado indexada*.

- Si las relaciones están ordenadas, sería deseable una *reunión por mezcla*. Además, podría ser útil ordenar una relación antes que calcular una reunión (para permitir el uso de una estrategia de reunión por mezcla).
- El algoritmo de *reunión por asociación* divide la relación en varias particiones, de tal manera que cada partición de una de las relaciones quepa en memoria. La división se lleva a cabo con una función de asociación en los atributos de la reunión, de tal modo que los pares de particiones correspondientes se puedan reunir independientemente.
- La eliminación de duplicados, la proyección, las operaciones de conjuntos (unión, intersección y diferencia) se pueden realizar mediante ordenación o asociación.
- Las operaciones de reunión externa se pueden implementar como extensiones simples de los algoritmos de reunión.
- Las técnicas de asociación y ordenación son duales, en el sentido en que muchas operaciones como la eliminación de duplicados, agregación, reuniones y reuniones externas que se pueden implementar mediante asociación también se pueden implementar por ordenación, y viceversa; es decir, cualquier operación que se pueda implementar con ordenación también puede serlo por asociación.
- Una expresión se puede evaluar mediante materialización, donde el sistema calcula el resultado de cada subexpresión y lo almacena en disco, y después lo usa para calcular el resultado de la expresión padre.
- El encauzamiento ayuda a evitar la escritura en disco de los resultados de muchas subexpresiones usando los resultados de la expresión padre incluso si se estuviesen generando.

## Términos de repaso

- Procesamiento de consultas.
- Evaluación de primitivas.
- Plan de ejecución de consultas.
- Plan de evaluación de consultas.
- Motor de ejecución de consultas.
- Medidas del coste de las consultas.
- E/S secuencial.
- E/S paralela.
- Exploración de archivos.
- Búsqueda lineal.
- Búsqueda binaria.
- Selecciones usando índices.
- Rutas de acceso.
- Exploración de índices.
- Selección conjuntiva.
- Selección disyuntiva.
- Índice compuesto.
- Intersección de identificadores.
- Ordenación externa.
- Ordenación–mezcla externa.
- Secuencias.
- Mezcla de  $n$  vías.
- Equirreunión.
- Reunión en bucle anidado.
- Reunión en bucle anidado por bloques.
- Reunión en bucle anidado indexada.
- Reunión por mezcla.
- Reunión por ordenación–mezcla.
- Reunión por mezcla híbrida.
- Reunión por asociación.
  - Construir.
  - Prueba.
  - Entrada de construcción.
  - Entrada de prueba.
  - División recursiva.
  - Desbordamiento de la tabla de asociación.
  - Sesgo.
  - Factor de escape.
  - Resolución del desbordamiento.
  - Evitación del desbordamiento.
- Reunión por asociación híbrida.
- Árbol de operadores.
- Evaluación materializada.
- Memoria intermedia doble.
- Evaluación encauzada.

- Cauce bajo demanda  
(perezoso, extracción).
- Cauce por productor  
(impaciente, inserción).
- Iterador.
- Reunión encauzada.

## Ejercicios prácticos

13.1 Considérese la siguiente consulta SQL para la base de datos bancaria:

```
select T.nombre_sucursal
from sucursal T, sucursal S
where T.activos > S.activos and S.ciudad_sucursal = "Arganzuela"
```

Escríbase una expresión del álgebra relacional equivalente a la dada que sea más eficiente. Justifíquese la elección.

- 13.2 Supóngase (para simplificar este ejercicio) que solamente cabe una tupla en un bloque y que la memoria puede contener como máximo tres marcos de página. Muéstrense las secuencias creadas en cada ciclo del algoritmo de ordenación—mezcla cuando se aplica para ordenar el primer atributo de las siguientes tuplas: (canguro, 17), (ualabí, 21), (emu, 1), (wombat, 13), (ornitorrinco, 3), (león, 8), (jabalí, 4), (cebra, 11), (koala, 6), (hiena, 9), (cálao, 2), (babuino, 12).
- 13.3 Dadas las relaciones  $r_1(A, B, C)$  y  $r_2(C, D, E)$  con las siguientes propiedades:  $r_1$  tiene 20.000 tuplas,  $r_2$  tiene 45.000 tuplas, en cada bloque caben, como máximo, 25 tuplas de  $r_1$  o 30 tuplas de  $r_2$ . Estímese el número de transferencias de bloques y de búsquedas necesarias utilizando las siguientes estrategias para la reunión  $r_1 \bowtie r_2$ :
- Reunión en bucle anidado.
  - Reunión en bucle anidado por bloques.
  - Reunión por mezcla.
  - Reunión por asociación.
- 13.4 El algoritmo de reunión en bucle anidado indexada descrito en el Apartado 13.5.3 puede ser ineficiente si el índice fuera secundario y hubiese varias tuplas con el mismo valor en los atributos de la reunión. ¿Por qué es ineficiente? Describáse una forma de reducir el coste de recuperar las tuplas de la relación más interna utilizando ordenación. ¿Bajo qué condiciones sería este algoritmo más eficiente que la reunión por mezcla híbrida?
- 13.5 Sean  $r$  y  $s$  dos relaciones sin índices que no están ordenadas. Suponiendo una memoria infinita ¿cuál es la manera más económica (en términos de operaciones de E/S) para calcular  $r \bowtie s$ ? ¿Cuánta memoria se necesita en este algoritmo?
- 13.6 Supóngase que hay un índice de árbol B<sup>+</sup> disponible en *ciudad\_sucursal* de la relación *sucursal* y que no hay más índices. ¿Cuál sería el mejor modo de manejar las siguientes selecciones con negaciones?
- $\sigma_{\neg(ciudad\_sucursal < "Arganzuela)}(sucursal)$
  - $\sigma_{\neg(ciudad\_sucursal = "Arganzuela)}(sucursal)$
  - $\sigma_{\neg(ciudad\_sucursal < "Arganzuela) \vee activos < 5000}(sucursal)$
- 13.7 Escríbase el pseudocódigo para un iterador que implemente la reunión en bucle anidado indexada, donde la relación externa esté encauzada. Defínanse las funciones iteradoras estándar *open()*, *next()* y *close()*. Muéstrese la información de estado del iterador que se debe guardar entre las llamadas.
- 13.8 Diseñense algoritmos basados en ordenación y asociación para el cálculo de la operación división.
- 13.9 ¿Cuál es el efecto sobre el coste de la mezcla de secuencias si el número de bloques de memoria intermedia se incrementa en cada ciclo mientras se mantiene la memoria total disponible para la memoria intermedia de las secuencias?

## Ejercicios

- 13.10** ¿Por qué no hay que obligar a los usuarios a que elijan explícitamente una estrategia de procesamiento de la consulta? ¿Hay casos en los que es deseable que los usuarios sepan el coste de las distintas estrategias posibles? Razónese la respuesta.
- 13.11** Diséñese una variante del algoritmo híbrido de reunión por mezcla para el caso en el que las dos relaciones no están ordenadas según el orden físico de almacenamiento, pero ambas tienen un índice secundario ordenado en los atributos de la reunión.
- 13.12** Estímese el número de accesos a bloques necesitados por la solución del Ejercicio 13.11 para  $r_1 \bowtie r_2$ , donde  $r_1$  y  $r_2$  son como las relaciones definidas en el Ejercicio 13.3.
- 13.13** El algoritmo de reunión por asociación descrito en el Apartado 13.5.5 calcula la reunión natural de dos relaciones. Describáse cómo extender el algoritmo de reunión por asociación para calcular la reunión externa por la izquierda, la reunión externa por la derecha y la reunión externa completa. *Sugerencia:* se puede mantener información adicional con cada tupla en el índice asociativo para detectar si alguna tupla en la relación de prueba concuerda con alguna tupla del índice asociativo. Compruébese el algoritmo con las relaciones *cliente* e *impositor*.
- 13.14** Escríbase pseudocódigo para un iterador que implemente una versión del algoritmo ordenación–mezcla donde el resultado de la mezcla final se encauce a los consumidores. El pseudocódigo debe definir las funciones estándar del iterador *abrir ()*, *siguiente ()* y *cerrar ()*. Muéstrese la información de estado que el iterador debe mantener entre cada llamada.
- 13.15** Se usa encauzamiento para evitar escribir resultados intermedios a disco. Supóngase que se necesita ordenar una relación  $r$  usando ordenación–mezcla y reuniendo por mezcla el resultado con una relación  $s$  previamente ordenada.
- Describáse cómo se puede encauzar la salida de la ordenación de  $r$  con la reunión por mezcla sin escribir a disco.
  - La misma idea es aplicable incluso si ambas entradas a la reunión por mezcla son las salidas de las operaciones de ordenación–mezcla. Sin embargo, la memoria disponible tiene que compartirse entre las dos operaciones de mezcla (el propio algoritmo de reunión por mezcla necesita muy poca memoria). ¿Cuál es el efecto de tener que compartir la memoria sobre el coste de cada operación de ordenación–mezcla?
- 13.16** Supóngase que hay que calcular  ${}_A\mathcal{G}_{sum(C)}(r)$  y  ${}_{A,B}\mathcal{G}_{sum(C)}(r)$ . Describáse cómo calcular ambas juntas usando una única ordenación de  $r$ .

## Notas bibliográficas

Todos los procesadores de consultas deben analizar instrucciones del lenguaje de consulta y deben traducirlas a su formato interno. El análisis de los lenguajes de consultas difiere poco del análisis de los lenguajes de programación tradicionales. La mayoría de los libros sobre compiladores, como Aho et al. [1986] y Tremblay y Sorenson [1985], tratan las principales técnicas de análisis y presentan la optimización desde el punto de vista de los lenguajes de programación.

Graefe [1993] presenta una excelente revisión de las técnicas de evaluación de consultas.

Knuth [1973] presenta un excelente descripción de algoritmos de ordenación externa, incluyendo una optimización denominada *selección de reemplazamiento* que puede originar secuencias iniciales que son (en media) el doble del tamaño de la memoria. Estudios más recientes en Nyberg et al. [1995] han demostrado que debido al mal comportamiento de la caché del procesador, la selección de reemplazamiento se comporta peor que quicksort en memoria para la generación de secuencias, eliminando las ventajas de la generación de secuencias más grandes. Nyberg et al. [1995] presenta una algoritmo eficiente de ordenación externa que considera los efectos de la caché del procesador. Los algoritmos de evaluación de consultas que consideran los efectos de la caché se han estudiado ampliamente; véase, por ejemplo, Harizopoulos y Ailamaki [2004].

De acuerdo con estudios del rendimiento realizados a mediados de los años setenta del siglo veinte, los sistemas de bases de datos de esa época utilizaban solamente reunión en bucle anidado y reunión por mezcla. Estos estudios, que estuvieron relacionados con el desarrollo de System R, determinaron que tanto la reunión en bucle anidado como la reunión por mezcla casi siempre proporcionaban el método de reunión óptimo (Blasgen y Eswaran [1976]; por tanto, estos son los dos únicos algoritmos de reunión implementados en System R. Sin embargo, el estudio de System R no incluyó el análisis de los algoritmos de reunión por asociación. Actualmente, estos algoritmos se consideran muy eficientes y se usan mucho.

Los algoritmos de reunión por asociación se desarrollaron inicialmente para sistemas de bases de datos paralelos. La técnica de reunión por asociación se describe en Kitsuregawa et al. [1983] y en Shapiro [1986] se describen extensiones incluyendo la reunión por asociación híbrida. Resultados más recientes de Zeller y Gray [1990] y de Davison y Graefe [1994] describen técnicas de reunión por asociación que se pueden adaptar a la memoria disponible, que es importante en sistemas donde se pueden ejecutar a la vez varias consultas. Graefe et al. [1998] describe el uso en SQL Server de Microsoft de las reuniones por asociación y los *equipos asociados*, que permiten el encauzamiento de las reuniones por asociación usando la misma división para todas las reuniones por asociación en una secuencia encauzada.

# Índice

- 1FN, véase forma normal, primera  
2FN, véase forma normal, segunda  
3FN, véase forma normal, tercera  
4FN, véase forma normal, cuarta
- a priori, 622
- ABD, véase administrador de bases de datos
- abstracción, 605
- abstracción de datos, 4
- nivel de vistas, 5
  - nivel físico, 5
  - nivel lógico, 5
- Access de Microsoft, 59
- ACID, 508, 777, 780
- Active Directory, 290
- Active Server Pages, 270
- Active Server Pages.NET, 272
- actualizaciones, 55, 87
- ad hoc, 743
- administrador de bases de datos, 22
- ADO (ActiveX Data Objects), 117, 747, 905
- ADO.NET, 117
- AES (Advanced Encryption Standard), 286
- Agarwal, Sameet, 885
- agregación, 196
- funciones, 49, 73, 75
  - binarias, 608
  - operación, 50
  - paralela, 684
- agrupación de conexiones, 273
- agrupaciones, 615, 660
- agrupamiento, 623
- aglomerativo, 623
  - divisivo, 623
  - jerárquico, 623
- Ailamaki, Anastassia, 807
- aislamiento, 508, 509
- AIX, 859
- ajuste de la curva, 620
- ajuste del rendimiento, 733
- álgebra relacional, 36
- expresión, 38, 715
- algoritmo
- de acoso, 713
  - de elección, 713, 724
  - de grafos, 644
  - de programación dinámica, 489
  - de recuperación, 701
  - del ascensor, 373
  - híbrido de reunión por mezcla, 457
- impaciente, 616
- Rijndael, 286
- alias, 696, 721
- de tipos, 102, 103
- all privileges, 111
- almacenamiento, 367
- óptico, 368
  - conectado en red, 372
  - distribuido de datos, 694
  - en cinta, 368
  - en conexión, 369
  - en discos magnéticos, 368
  - estable, 568
  - nativo, 353
  - no volátil, 369, 568, 584
  - primario, 369
  - secundario, 369
  - seguro, 287
  - sin conexión, 369
  - terciario, 369, 382
  - volátil, 369, 568
- almacenes de datos, 602, 612
- arquitectura dirigida por el destino, 613
- los orígenes, 613
- alta disponibilidad, 567
- Amazon, 357
- ámbito, 313
- Ampex, exploración helicoidal, 383
- ampliabilidad, 658
- de transacciones, 658
  - lineal, 658
  - por lotes, 658
  - sublineal, 658
- análisis estadístico, 602
- anfitriones móviles, 768
- anidamiento, 312
- anidar, 332
- subconsultas, 76
- ANSI (American National Standards Institute), 61, 745, 809, 891
- Apache
- DB, 328
  - proyecto Jakarta, 270
  - sistema Tomcat, 297
- aplicaciones
- multisistema, 781
- árboles, 340
- B+, véase índices, árbol B+
  - B, véase índices, árbol B
- binario, 410
- completo, 500
- cuadráticos, 762
- PR, 762
  - regionales, 762
- de operadores, 466, 678
- en memoria, 410
- equilibrado, 408
- k-d, 761
  - k-d B, 761
- R, véase índices, árbol R
- archivos, 386
- cabecera, 388
  - estructura, 390
  - organización, 373
    - asociativa (hash), 390
    - en montículos, 390
    - secuencial, 390
  - reorganizar, 391
  - secuenciales, 390
    - fragmentados, 374
    - indexados, 402
  - secuenciales indexados, 436
- ARIES, 566, 588, 597, 803, 824, 859, 879, 902
- Armstrong, axiomas, 232
- completos, 232
  - correctos, 232
- Arpanet, 667
- arquitectura
- de dos capas, 20
  - de memoria no uniforme, 663
  - de tres capas, 20
  - disco compartido, 660
  - jerárquica, 660, 662
  - memoria compartida, 660
  - paralela de bases de datos, 660
  - sin compartimiento, 660
- Arquitectura común de agente para solicitudes de objetos, 747
- ASCII, 717
- aserto, 110
- asignación, operación, 48
- asistentes para el ajuste, 738
- asociación, 421
- abierta, 424
  - cerrada, 424
  - dinámica, 426
  - esquemas, 431
  - extensible, 426, 430
  - función, 401
  - lineal, 431
  - reglas, 621
- asociaciones, 615, 621

- asociativa, propiedad, 46, 478  
 ASP (Active Server Pages), 260, 270, 272  
 ASP.NET, 272  
 asunción de un rol único, 249  
 atómico, 223  
 ATA (AT Attachment), 371  
 ataques  
     de diccionario, 287  
     de personas intermedias, 788  
 atasco, 423  
 atomicidad, 18, 507, 508, 512, 567, 571  
     ante fallos, 783  
 atributos, 13, 29, 172, 334  
     autorreferencial, 313  
     compuesto, 175, 304  
     de dimensión, 603  
     de medida, 603  
     de partición, 617  
     derivado, 176  
     descriptivo, 174  
     determinado funcionalmente, 233  
     determinantes, 618  
     monovalorado, 175  
     multivalorado, 175  
     raros, 235  
     simple, 175  
     valor, 172  
 aumento, 522  
 autenticar, 288  
 autodocumentado, 331  
 autonomía, 717  
     local, 664  
 autoridad, 637  
 autorización, 9, 62, 111  
     de actualización, 9  
     de eliminación, 9  
     de inserción, 9  
     de lectura, 9  
     en el nivel de las filas, 285  
     read, 284  
     update, 284  
 axiomas, 232  
 ayuda a la toma de decisiones, 602, 673, 742, 743
- B+, árbol, *véase* índices, árbol B+  
 B, árbol, *véase* índices, árbol B  
 bases de datos, 1  
     CAD, 323  
     con memoria intermedia, 582  
     de diseño, 757  
     distribuidas  
         heterogéneas, 693  
         homogéneas, 693  
     en memoria principal, 788  
     móviles, 754  
     multimedia, 765  
     orientadas a objetos, 301, 322  
         basadas en lenguajes de programación persistentes, 323  
     relacionales  
         diseño, 219  
         orientadas a objetos, 322  
     relacionales basadas en objetos, 301  
     temporales, 754
- virtual, 718  
     privada, 284  
 batidores Web, 641  
 Bayes  
     lógica ingenua, 920  
     teorema, 620  
 BEA, 270, 777  
 biblioteca de etiquetas, 271  
 big-endian, 718  
 billete, 798  
 bits de sal, 287  
 Blakeley, José A., 885  
 blob, 104  
 bloqueos, 522, 529, 700  
     bajada, 535  
     compartido, 544  
     compatibilidad, 530  
     comunicación, 657  
     con límite de tiempo, 550  
     concesión, 529, 533  
     conversiones, 535  
     de dos fases, 792  
     de la siguiente clave, 559  
     de valores clave, 559  
     del índice, 555  
     en modo compartido, 529  
     en modo exclusivo, 529  
     exclusivo, 544  
     explícito, 544  
     expropiación, 549  
     función de compatibilidad, 529  
     gestor, 536  
         único, 704  
         distribuido, 704  
     implícito, 544  
     liberación, 657  
     modos  
         intencional, 545  
         intencional-compartido, 545  
         intencional-exclusivo, 545  
         intencional-exclusivo y compartido, 545  
     protocolos, 560  
         con árboles B enlazados, 558  
         de árbol, 538  
         de dos fases, 533, 544  
         de dos fases multiversión, 548  
         de granularidad múltiple, 546  
         definición, 532  
         del índice, 555  
         del cangrejo, 557  
         estricto de dos fases, 534  
         riguroso de dos fases, 534  
         solicitud, 529  
         subida, 535  
         tabla, 536  
     bloques, 373, 383, 570  
     clavados, 384  
     de memoria intermedia, 570  
     físicos, 570  
 Bluetooth, 769  
 bolsas, 787  
 borrado, 54, 84  
 borrar, instrucción, 553, 555  
 brazo del disco, 370
- buffer, *véase* memoria intermedia  
 bus, 660  
 búsqueda, 558  
     binaria, 446  
     difusa, 613  
     lineal, 446
- C, 11, 62, 112  
 DB2, 863  
     monitores de teleprocesamiento, 779  
     PostgreSQL, 809  
 C++, 11, 62, 316, 319, 507  
     monitores de teleprocesamiento, 779  
 C2F, *véase* protocolos, de compromiso, de dos fases  
 C3F, *véase* protocolos, de compromiso, de tres fases  
 cabeza, 153  
     de lectura y escritura, 370  
 caché, 367  
     alojar, 657  
     coherencia, 657, 677  
 CAD (Computer Aided Design), 756, 757  
 cadena de desbordamiento, 424  
 cadenas de caracteres  
     operaciones, 69  
 caída del sistema, 567  
 caja límite, 762  
 cajones, 421  
     de desbordamiento, 424  
     desbordamiento, 423  
 cálculo relacional  
     de dominios, 141  
     de tuplas, 137  
 Call Level Interface, *véase* CLI (Call Level Interface)  
 cambiadores automáticos, 368  
     de cintas, 383  
     de discos, 382  
 cambio de contexto, 778  
 caminos de acceso, 447  
 carga, 614  
     carga de trabajo, 498, 738  
 case, 124  
 casos, 621  
 catálogos, 104, 117, 120, 356  
     del sistema, 393  
 cauce, 466  
 CD (Compact Disk), 368  
 celda, 768  
 celular, *véase* teléfonos móviles  
 centroide, 623  
 certificados digitales, 289, 788  
 CGI (Common Gateway Interface), *véase* interfaz, de pasarela común  
 ChemML, 356  
 ciclos falsos, 710  
 CICS, 777  
 cierre, 226, 232, 234, 246  
     transitivo, 127, 160, 160  
 cifrado  
     clave pública, 286  
     clave privada, 286  
 CIFS, 372  
 cilindro, 370

- cintas magnéticas, 382  
 círculos, 756  
 clases de objetos, 720  
 clasificación, 616  
   densa, 98  
   por popularidad, 635  
   por prestigio, 635  
 clasificadores  
   bayesianos, 620, 620  
   ingenuos, 620  
   de árboles de decisión, 616  
   de redes neuronales, 620  
 clave de búsqueda, 390  
 claves, 178  
   candidata, 34, 178  
   de búsqueda, 402  
   externa, 35, 107, 109, 279  
   primaria, 34, 63, 178  
 claves de búsqueda  
   accesos bajo varias, 418  
   compuesta, 419  
 CLI (Call Level Interface), 117, 746  
   normas, 745  
 clientes, 20  
 clob, 104  
 CLR (Common Language Runtime), 125, 908, 909  
 CLR .NET, 908  
 Cobol, 11, 62, 112  
 CODASYL de DBTG, 745  
 Codd, E. F., 23, 59, 257  
 códigos  
   de corrección de errores tipo memoria, 377  
   de Reed-Solomon, 380  
 coerción, 102  
 coincidencia, 347  
 cola duradera, 780  
 ColdFusion, véase lenguajes, de marcas, de ColdFusion  
 COM (Component Object Model), 908  
 Common Object Request Broker  
   Architecture, véase CORBA  
 componentes  
   control de concurrencia, 510, 516  
   gestión de recuperaciones, 509  
   gestión de transacciones, 18, 509  
 compresión  
   de la carga de trabajo, 739  
   del prefijo, 417  
 comprobación  
   de la secuencialidad, 522  
   de validación, 543  
 compromiso, véase protocolos, de compromiso  
 concepto, 639  
 concreción, 605  
 condiciones  
   de excepción, 124  
   de partición, 617  
 conexión  
   continua, 667  
   discontinua, 667  
 confianza, 621  
 conflicto, 517  
 conjunto de entidades, 13, 171  
   débiles, 189, 189, 203  
   de nivel inferior, 193  
   de nivel superior, 193  
   fuertes, 189, 203  
   identificadoras, 189  
   propietarias, 189  
 conjunto de relaciones, 13, 173, 179, 200  
   binario, 175  
   grado, 175  
   recursivo, 173  
 conjuntos  
   comparación, 77  
   de formación, 616  
   de resultados actualizables, 120  
   de valores, 175  
   grandes de elementos, 622  
   operaciones, 71  
 conmutativa, propiedad, 477  
 connect by, 834  
 Consejo para el rendimiento del procesamiento de las transacciones, 742  
 consenso de quórum, 706  
 conserva las dependencias, 229  
 consistencia, 18, 507, 508, 770  
   de grado dos, 555  
   de los datos, 105, 513, 581  
 consistente en cuanto a operaciones, 588  
 consulta gráfica mediante ejemplos, 150  
 consultas, 410  
   básica, 128  
   basada en conceptos, 639  
   concreta, 674  
   de los primeros K, 503  
   de proximidad, 760  
   de rango, 674  
   de vecino más próximo, 760  
   definición, 8  
   dependiente de la ubicación, 768  
   espacial, 760  
   monótona, 129  
   motor de ejecución, 444  
   optimización, 475, 686  
     basada en costes, 476  
     con tableau, 503  
     de actualizaciones, 503  
     de agregación, 502  
     de los primeros K, 503  
     de multiconsultas, 503  
     paramétrica, 502  
     semántica, 503  
   plan de ejecución, 444  
   plan de evaluación, 444  
   procesador, 22  
   procesamiento, 443  
     distribuido, 714  
     técnicas, 789  
   procesamiento distribuido, 769  
   recursiva, 128  
   regional, 760  
     sobre una relación, 144  
     sobre varias relaciones, 146  
   contador lógico, 540  
 contaminación de los motores de búsqueda, 638  
 contenido de información, 618  
 contexto, 333  
 control de admisión, 766  
 control de concurrencia, 701  
   esquema, 543, 579  
   esquemas, 514, 522, 529  
   multiversión, 547, 793  
   optimista, 544  
   subsistema, 386  
 control de transacciones, 62  
 controlador de disco, 371  
 cookie, 266  
 coordinador  
   de transacciones, 697  
   suplente, 713  
 copia en la sombra, 512  
 copia principal, 694, 705  
 copo de nieve, esquema, 614  
 CORBA, 747, 749  
 corrección fuerte, 797  
 correlaciones, 622  
 corresponden, 308  
 correspondencia de cardinalidades, 14, 177, 200  
 corte, 605  
   de cubos, 605  
 costes, 714  
   de inicio, 659  
 crawlers, véase batidores Web  
 creación de imágenes, 375, 377  
 creador de mercado, 787  
 cross join, 94  
 Crystal Reports, 261  
 cuadrícula de diseño, 150  
 cuadro de condiciones, 147, 147  
 cubo, 125  
 cubos de datos, 604  
 cuellos de botella, 733  
 cuerpo, 153  
 cume\_dist, 610  
 current\_date, 102  
 current\_time, 102  
 D'Hers, Thierry, 885  
 DAT (Digital Audio Tape), 383  
 Data Encryption Standard, 286  
 data mining, véase minería de datos  
 Data Universal Numbering System, 747  
 data warehouses, véase almacenes de datos  
 Database Task Group, 745  
 Datalog, 27, 137, 151, 162  
   programa, 153  
   recursividad, 158  
   reglas, 153  
 DataSet, 747  
 datos  
   archivados, 368  
   CAD, 756  
   de difusión, 770  
   de medios continuos, 754, 765  
   espaciales, 753  
   fragmentados, 353

- datos (*cont.*)  
 geográficos, 753, 756, 758, 759  
 aplicaciones, 759  
 globales, 797  
 isócronos, 765  
 locales, 797  
 medios continuos, 766  
 multidimensionales, 603  
 multimedia, 754  
 formatos, 766  
 por líneas, 758  
 temporales, 251, 753  
 válidos, 251  
 vectoriales, 759
- DB2 de IBM, 23, 59
- DB2 Universal Database de IBM, 859
- DBA, véase administrador de bases de datos
- DBC de Teradata, 680, 687
- DBTG, véase Database Task Group
- DEC (Digital Equipment Corporation), 671
- definición de tipos de documentos, 335
- delete, 64, 887
- depende  
 de, 154  
 directamente de, 154  
 indirectamente de, 154
- dependencias  
 de compromiso, 539  
 de reunión, 248  
 de subconjuntos, 107  
 del valor, 798  
 existencial, 189  
 parcial, 255  
 que generan igualdades, 245  
 que generan tuplas, 245  
 transitivas, 230
- dependencias funcionales, 14, 222, 225, 231  
 cumple, 225  
 implicada lógicamente, 232  
 temporales, 251, 755  
 triviales, 226
- dependencias multivaloradas, 244-246  
 trivial, 245
- DES, 286
- desanidamiento, 311
- desbordamiento de una tabla de asociación, 460
- descomposiciones, 224  
 con pérdidas, 223, 237  
 que conservan las dependencias, 238  
 sin pérdidas, 223, 237
- desconexiones, 770
- descorrelación, 494
- descripción, descubrimiento e integración universales, 357
- desduplicación, 613
- desnormalización, 250
- desviación, 622
- detección de fallos, 592
- diagrama E-R, 180, 200
- diagramas de esquema, 35
- diccionarios de datos, 9, 393, 394, 837, 842
- diferencia de conjuntos, operación, 39, 464
- diferencial, 495
- DigiCash, 788
- Digital Subscriber Line, 667
- dimension, 843
- directorio, 643, 644
- Directorio Activo, 290
- discos  
 ópticos, 382  
 compartido, 662  
 con imagen, 569  
 de registro histórico, 374  
 de video digital, 368  
 digital versátil, 368  
 fallo, 567  
 magnéticos, 370  
 planificación, 373  
 salida forzada de bloques, 384
- discriminante, 189
- diseño conceptual, 12, 170
- disparador, 273
- disponibilidad, 710  
 elevada, 591, 710
- dispositivo cabeza-disco, 370
- distinct types, 102
- distribución de datos, 376  
 en el nivel de bits, 376  
 en el nivel de bloques, 376
- distribuidor, 907
- dividir, 411
- división, 558  
 cuadrática, 764  
 de la red, 701  
 estrategias, 674  
 horizontal, 674  
 operación, 46  
 por asociación, 681  
 por rangos, 681  
 recursiva, 460  
 sesgada, 460  
 sesgo, 676
- DLT (Digital Linear Tape), 383
- documentos sin estructurar, 632
- DOM (Document Object Model), 265, 349
- domain type, 103
- domiciliación, 613
- dominios, 29, 63, 140, 175  
 atómicos, 30, 302
- drop trigger, 276
- DSL, 667
- DTD (Document Type Definition), 335
- DUNS, 747
- duplicación  
 instantánea, 907  
 por mezcla, 907  
 transaccional, 907
- durabilidad, 18, 508, 509, 512, 567  
 grados  
 dos muy seguro, 593  
 dos seguro, 593  
 uno seguro, 593
- DVD (Digital Video Disk), 368
- EBCDIC, 717
- ejecuciones no secuenciales, 792
- ejemplares, 6, 154  
 básico de una regla, 154  
 de la base de datos, 31  
 de la relación, 31, 173
- ejemplos de formación, 616
- elección, 711
- elemento, 332, 656
- eliminación de duplicados, 145  
 paralela, 684
- elipses, 756
- encauzamiento  
 bajo demanda, 468  
 por los productores, 468
- Encina, 777, 781
- enfoque incremental, 749
- entidad, 171  
 de procesamiento, 781
- entornos distribuidos, 704
- entradas, 720  
 de construcción, 459  
 de prueba, 459
- envolturas, 718, 748
- equipos asociados, 474
- equirreunión, 452
- equivalentes, 476
- ERP, 785
- error  
 del sistema, 567  
 lógico, 567
- es/son, 639
- escribir, operación, 793
- escrituras  
 a ciegas, 520  
 externas observables, 511
- espacio de intercambio, 583
- espacio de nombres, 334  
 predeterminado, 334
- espacios de tablas, 837
- especialización, 190, 191, 308  
 parcial, 194  
 total, 194
- esperar, 530
- esperar--morir, 549
- esqueletos de tablas, 144, 149  
 resultado, 149
- esquemas  
 único de relación, 149  
 de la base de datos, 31  
 de la relación, 31  
 definición, 6, 63  
 físico, 6  
 lógico, 6
- estabilidad del cursor, 556
- estaciones para el soporte de movilidad, 768
- estadísticas y estimación del coste, 502
- estado de ejecución, 468, 783
- estados  
 de terminación  
 aceptables, 783  
 no aceptables, 783
- inconsistente, 509
- preparado, 699
- estrategia  
 de extracción inmediata, 385

- estrategia (*cont.*)
  - de gestión de la memoria intermedia, 385
  - de semirreunión, 716
  - de sustitución, 384
  - más recientemente utilizado, 385
- estrella, esquema, 614
- estructura
  - de índice ordenado, 420
  - de páginas con ranuras, 389, 389
- etiquetas, 329
- evaluación
  - correlacionada, 493
  - encauzada, 467
  - materializada, 467
- evento-condición-acción, 274
- evitación del desbordamiento, 461
- except, 71
- excepto, operación, 72
- exceso de ajuste, 619
- exclusión mutua, 655
- expansión de vistas, 83, 156
- exploración
  - de la relación, 490, 674
  - del índice, 447, 490
  - sólo del índice, 490
- explorador de archivo, 446
- exposición de datos no comprometidos, 791
- expresión de ruta, 341
- expresiones
  - de camino, 315
- extensiones de clases, 319
- extensiones persistentes, 323
- extracción, 468, 614
- extracción de información, sistemas, 642
- factor de escape, 461
- falla
  - durante una transferencia de datos, 569
  - en el sistema, 512
- falso negativo, 640
- falso positivo, 640
- fantasma, fenómeno, 554, 559
- fase de diseño
  - físico, 12, 170
  - lógico, 12, 170
- fases
  - crecimiento, 533
  - decrecimiento, 533
  - deshacer, 587
  - escritura, 543
  - lectura, 543
  - rehacer, 587
  - validación, 543
- fiabilidad, 375
- Fibre Channel, 373
- fila, 30
- filas de ejemplo, 144
- FireWire, 371
- firmas digitales, 288
- Flashback, 850
- fluxos de trabajo, 703, 783
  - definición, 781
  - ejecución, 782
- especificación, 782, 783
  - insegura, 785
  - estado, 783
- recuperación, 785
- sistema gestor, 784
  - arquitectura centralizada, 784
  - arquitectura completamente distribuida, 784
  - arquitectura parcialmente distribuida, 784
- FLWOR, 343
- FNBC, véase forma normal, de Boyce-Codd
- FNDC, véase forma normal, de dominios y claves
- foreign key, 107
- forma normal, 219
  - cuarta, 244, 246
  - de Boyce-Codd, 226
  - de dominios y claves, 248
  - de reunión por proyección, 248
  - primera, 224, 302
  - quinta, 248
  - segunda, 229, 248, 255
  - tercera, 229
  - algoritmo de síntesis, 242
- Forms de Oracle, 260
- Fortran, 62, 112
- FoxPro, 59
- fragmentación
  - de los datos, 695
  - horizontal, 695, 695
  - vertical, 695, 695
- fragmentos y réplicas, 681, 683
- frecuencia del término, 633
- frecuencia inversa de los documentos, 633
- from, 67, 311
- función
  - de asociación, 421
- funciones
  - constructoras, 306
  - de agregación
    - no-descomponibles, 607
  - de tabla, 121
- fusión, 558
- fusionar, 411
- ganancia de información, 618
- ganancia de velocidad, 658
  - lineal, 658
  - sublineal, 658
- generación impaciente de tuplas, 468
- generación interactiva de índices, 688
- generalización, 191, 308
  - de solapamiento, 194
  - parcial, 194
  - sobre la condición de disjunción, 194
  - total, 194
- gestor
  - de bloqueos, véase bloqueos, gestor
  - de colas, 780
  - de control de concurrencia, 18
  - de la memoria intermedia, 384
  - de recursos, 780
  - de transacciones, 697
- getConnection, 117
- Global Trade Item Number, 747
- GNU, 808
- Google, 357
- google.com, 266
- GPS (Global Positioning System), 759
- GQBE, 150
- grafo
  - de autorización, 280
  - de espera, 551
  - de la base de datos, 538
  - de precedencia, 522, 525
  - dirigido acíclico, 538
  - global de espera, 709
  - local de espera, 708
- gran explosión, enfoque, 749
- grant, 111, 280
- granularidad, 544
  - múltiple, 544
- Graphical Query-By-Example (GQBE), 150
- Grupo de administración de objetos, 747
- Grupo de gestión de bases de datos de objetos, 747
- grupos, 50
- GTIN, 747
- guiones en el lado del servidor, 270
- hebra, 654
- hecho, 153
- herencia
  - única, 193
  - de los atributos, 193
  - múltiple, 193, 307
- herir-esperar, 549
- heurísticas, 502
- Himalaya de Compaq, 687
- Hinson, Gerald, 885
- hipercubo, 660
- hipervínculos, 262
- histograma, 483, 676
  - de equianchura, 483
  - de equiprofundidad, 483
- hojas de estilo, 264, 347
  - en cascada, 264
- homónimos, 638
- hora universal coordinada, 755
- HP-UX, 859
- HTML (Hyper-Text Markup Language), 329
- HTML-DB, 260
- HTTP (Hyper-Text Transfer Protocol), 263, 748
- HTTPS (Hyper-Text Transfer Protocol Secure), 289
- IDE (Integrated Drive Electronics), 371
- idempotente, 574
- identificador del elemento de datos, 572
- identificadores lógicos de fila, 839
- IDF (Inverse Document Frequency), 633
- IDL (Interface Description Language), 747
- id Tupla, 695

- IEEE (Institute of Electrical and Electronics Engineers), 745  
 inanición, 533, 550, 552  
 inconsistencia de los datos, 3  
 incrementar, 563  
 independencia  
     de recuperación, 591  
     física respecto de los datos, 6  
 indexación ordenada  
     esquemas, 431  
 índices, 401  
     árbol B, 417  
         algoritmo de control de concurrencia, 585  
         enlazados, 558  
     árbol B+, 408, 415, 431, 561  
         borrado, 411  
         inserción, 411  
         redistribuir, 412  
     árbol R, 420  
 asociativo, 401, 421, 425  
     construcción, 459  
     prueba, 459  
 compuesto, 449  
 con agrupación, 402  
 construcción ascendente, 439  
 datos espaciales, 761  
 de cobertura, 420  
 de función, 352  
 de ganancia de información, 618  
 de proyección, 441  
 denso, 403  
 disperso, 403  
 documentos, 639  
 entrada, 403  
 espacio adicional requerido, 402  
 GiST, 816, 824, 826  
 invertido, 639  
 mapas de bits, 433  
     de existencia, 434  
     intersección, 433  
 multinivel, 405  
 ordenado, 401  
 por capas de bits, 441  
 primario, 402  
 secuencial, 408, 414, 431  
 secundario, 402  
 selección, 498  
 sin agrupación, 402  
 tiempo  
     de acceso, 402  
     de borrado, 402  
     de inserción, 402  
     tipos de acceso, 402  
 inferir, 155  
 informática móvil, 768  
 información geométrica, 756  
 informar, 743  
 informes de invalidación, 771  
 Informix, 59  
 ingeniería inversa, 748  
 Ingres, 23, 59, 807  
 inicio de sesión único, 289  
 inserción, 55, 85, 468  
 inserción y borrado, 558
- insert, 64, 887  
 insertar, 553  
 instantáneas, 251, 854  
     actualizable, 854  
     sólo de lectura, 854  
 instrucciones atómicas, 655  
 int, 102  
 integridad, 61  
     referencial, 107  
 interactivamente, 688  
 interbloqueos, 532, 546, 548, 565  
     detección, 549, 551, 708  
         centralizada, 709  
         prevención, 549, 708  
         recuperación, 549, 552  
 intercambio de operadores, 687  
 intercambio en caliente, 381  
 interfaz  
     de nivel de llamada, 746  
     de pasarela común, 265  
     del gestor de recursos, 780  
     para programas de aplicación, 115  
 interferencia, 659, 742  
 Internet, 667  
 intersección, operación, 44, 72, 464  
 intersect, 71  
 interval, 755  
 Inverse Document Frequency (IDF), 633  
 IPv4, 812  
 IPv6, 812  
 ISO (International Organization for Standardization), 61, 720, 745  
 iterador, 468, 829
- J2EE (Java 2 Enterprise Edition), 270, 834  
 Jakobsson, Hakan, 833  
 Java, 11, 62, 112, 265, 507  
 Java Database Objects (JDO), 321  
 Java Server Pages, 270  
 Javascript, 265  
 JDBC (Java Database Connectivity), 11, 62, 104, 115, 117, 265, 316, 347, 653, 835  
 JDeveloper, 834  
 JDO, 321  
 jerarquías, 127, 193, 605  
     de clasificación, 643  
 JPEG (Joint Picture Experts Group), 766  
 JScript, 270, 747  
 JSP (Java Server Pages), 260  
 jukebox, 368, 382
- Kerberos, 289  
 Krishnamurthy, Sailesh, 807
- límite de tiempo, 550  
 línea de suscriptor digital, 667  
 LAN, 666  
 LDAP (Lightweight Directory Access Protocol), 719, 720  
 LDAP Data Interchange Format, 721  
 LDD, véase lenguajes, de definición de datos  
 LDIF, 721  
 Least Recently Used, 384, 385  
 leer, 508
- leer uno, escribir todos, 712  
 leer uno, escribir todos los disponibles, 712  
 leer, operación, 793  
 lenguajes  
     de almacenamiento y definición de datos, 8  
     de consultas, 8, 35  
         incorporado, 316  
         no procedimental, 137  
         temporales, 755  
     de definición de datos, 7, 8, 61, 62  
     de descripción de interfaces, 747  
     de guiones, 265  
         del lado del cliente, 265  
     de manipulación de datos, 7, 61  
     de marcas, 329  
         de ColdFusion, 270  
         de hipertexto, 329  
         extensible, 7, 329  
         inalámbrico, 769  
     de marcas de hipertexto, 262  
     de modelado unificado, 210  
     de programación  
         persistentes, 301, 316, 322  
     estándar generalizado de marcas, 329  
     no procedimentales, 36  
     procedimentales, 35  
 limpieza de los datos, 613  
 línea  
     poligonal, 756  
     quebrada, 756  
 Linux, 807, 859  
 lista libre, 388  
 lista-deshacer, 580  
 lista-rehacer, 580  
 literal  
     negativo, 153  
     positivo, 153  
 little-endian, 718  
 llamada a procedimientos remotos, 781  
 LMD, véase lenguajes, de manipulación de datos  
 Loader, 855  
 Local Area Network, 666  
 localtime, 102  
 lógica de negocio, 21  
 Lotus Notes, 668  
 LRU, 385
- MAC (Media Access Control), 812  
 Macromedia Flash, 265  
 Macromedia Shockwave, 265  
 malla, 660  
 manejadores, 124  
 máquina paralela  
     grano fino, 653, 657  
     grano grueso, 657  
     masivamente paralela, 657  
 marca, 329  
 marca-temporal-E, 540, 564  
 marca-temporal-L, 540  
 marcas temporales  
     esquema multiversión, 547  
 materializa, 466

- media armónica, 742  
mediadora, aplicación, 358  
mediadores, sistemas, 718  
medidas  
  de Gini, 617  
  de la entropía, 617  
mejor partición, 618  
memoria  
  compartida, 661, 661  
  flash, 367  
  intermedia, 384  
  de disco, 570  
  de escritura no volátil, 374  
  doble, 467  
  forzar la salida, 571  
  no volátil de acceso aleatorio, 374  
  principal, 367  
  virtual, 582  
    distribuida, 662  
menos recientemente utilizado, 384  
mensajería persistente, 665  
  implementación, 703  
mensajes persistentes, 702  
merge, 131  
metadatos, 9  
métodos, 305  
métrica compuesta  
  consultas por hora, 744  
  precio/rendimiento, 744  
mezcla de N vías, 450  
mezcla-purga, operación, 613  
microcomputadora, 767  
minería de datos, 18, 602, 615  
minería de texto, 624  
modelo  
  de árbol, 340  
  de alambres, 758  
  de datos  
    de red, 7  
    definición, 6  
    entidad-relación, 13, 171  
    jerárquico, 7  
    orientado a objetos, 15  
    relacional, 29, 673  
    relacional orientado a objetos, 16  
  de espacio vectorial, 634  
  de objetos componentes, 908  
  de objetos documento, 265, 349  
  de proceso por cliente, 777  
  de recorrido aleatorio, 636  
  de servidor único, 778  
  de simulación del rendimiento, 741  
  de varios servidores y un solo  
    encaminador, 779  
  de varios servidores y varios  
    encaminadores, 779  
modificaciones no comprometidas, 575  
módulo de almacenamiento persistente, 122  
momento de la transacción, 251  
monótona, 161  
monitores de procesamiento de  
  transacciones, 777  
Most Recently Used, 385  
MP3, 766
- MPEG (Moving Picture Experts Group), 766  
multiconjuntos, 49, 309  
multienhebramiento, 778  
multiprogramación, 514  
multitarea, 778  
MVS, 859  
Myers, Dirk, 885  
número de secuencia del registro histórico  
  (NSR), 588, 589  
NAS (Network Attached Storage), 372  
Netscape, 270  
NetWare, 778  
NFS (Network File System), 372  
niveles de consistencia  
  compromiso de lectura, 557  
  lectura repetible, 557  
  secuenciable, 556  
  sin compromiso de lectura, 557  
nodos, 340, 637, 663  
nombre distinguido, 720  
  relativo, 720  
normas, 744  
  801.11, 769  
  802.16, 769  
  anticipativas, 744  
  de cifrado avanzado, 286  
  de cifrado de datos, 286  
  de facto, 744  
  formales, 744  
  JDBC, 117, 745  
  ODBC, 745, 746  
  reaccionarias, 744  
  SQL, 435  
    opcionales, 890  
X/Open Distributed Transaction  
  Processing, 780  
notificación, 273  
Novell, 778  
NSRPágina, 589  
nulos, 15, 31, 176  
NUMA (Nonuniform Memory  
  Architecture), 663  
numeración de versiones, 771  
NVRAM (Nonvolatile Random-Access  
  Memory), véase memoria, no volátil  
  de acceso aleatorio
- ObjectStore, 319  
objeto hueco, 322  
Objetos de bases de datos de Java (JDO),  
  321  
ODBC (Open Database Connectivity), 11,  
  62, 115, 265, 316, 653, 904  
ODMG (Object Database Management  
  Group), 319, 320, 747  
OLE-DB, 746, 899  
OMA (Object Management Architecture),  
  747  
OMG (Object Management Group), 747  
Online Analytical Processing, 742  
Online Transaction Processing, 742  
ontologías, 639  
OOI, 744
- Open Directory Project, 645  
operaciones  
  coste de la evaluación paralela, 684  
  deshacer, 575, 576  
    física, 585  
    lógica, 585  
  entrada, 570  
  escribir, 571  
  lógicas, 585  
  leer, 571  
  rehacer, 574, 576  
    fisiológicas, 588  
  relacionales, 157  
  salida, 570  
operadores relacionales, 70  
optimización  
  de consultas, 445, 475  
    global, 718  
    heurística, 491  
optimizadores basados en el coste, 488  
Oracle, 2, 23, 59, 121, 285, 327, 352, 399,  
  492, 597, 603, 678, 740, 789, 833  
ORB (Object Request Broker), 747  
orden  
  de secuencialidad, 523  
  interesante, 490  
  lexicográfico, 419  
  más significativo, 718  
  menos significativo, 718  
ordenación  
  con división por rangos, 679  
  externa, 450  
  paralela, 679  
  por marcas temporales, 540  
  topológica, 523  
ordenación y mezcla externas paralelas,  
  679  
ordenación-mezcla externa, 450  
order by, 70  
organización de archivos en agrupaciones  
  de varias tablas, 390, 392  
OS X, 807  
OS/400, 859  
páginas amarillas, 719  
páginas blancas, 719  
páginas Web activas, 264  
Padmanabhan, Sriram, 859  
PageRank, 636  
paginación en la sombra, 796  
palabras clave, 631  
palabras de parada, 633  
Papadimitriou, Spiros, 807  
parámetros ajustables, 735  
paralelismo  
  de datos, 679  
  de encauzamiento, 685  
  de grano grueso, 652  
  en consultas, 678  
  en operaciones, 678, 679  
  entre consultas, 677  
  entre operaciones, 678, 685  
  independiente, 686

- paridad distribuida con bloques entrelazados, 379  
 parte-de, 639  
 particiones, 698  
     binarias, 618  
     múltiples, 618  
 participación, 173  
     parcial y total, 179  
 Pascal, 62, 112  
 pasos, 781  
 PATA (Parallel ATA), 371, 373  
 patrones descriptivos, 615  
 perezosa, generación, 468  
 periodo de validez, 251  
 Perl, 270, 807, 809, 817  
 persistencia de los objetos, 317  
 pertenencia  
     a conjuntos, 76  
     definida por el atributo, 193  
     definida por el usuario, 193  
     definida por la condición, 193  
 pestillos, 583  
 petabyte, 369  
 PHP, 270, 809  
 Pirzada, Vaqar, 885  
 pistas, 370  
 pivotaje, 605  
 PL/I, 62, 112  
 planes sólo de índices, 502  
 planificaciones, 514  
     consultas paralelas, 686  
     equivalente  
         en cuanto a conflictos, 518  
         en cuanto a vistas, 520  
     legal, 533  
     recuperable, 521, 521  
     secuenciable  
         en cuanto a conflictos, 519  
         en cuanto a vistas, 520  
     secuencial, 514  
     sin cascada, 521, 521  
 plantillas, 347  
 plato, 370  
 población, 621  
 polígonos, 756  
 posee, 189  
 PostgreSQL, 807  
 postmaster, 830  
 PowerBuilder de Sybase, 260  
 precede, 533  
 precisión, 640  
 predicción, 615  
 preextracción, 656  
 prefs, 266  
 preparada, 665  
 primitivas de evaluación, 444  
 privilegios, 111  
     concesión, 279  
     execute, 279  
     references, 279  
         de SQL, 281  
     usage, 279  
 problema de Halloween, 504  
 procesadores  
     local, 677  
     virtual, 676  
 procesamiento  
     en conexión  
         analítico, 742  
         de transacciones, 742  
 proceso  
     escritor de bases de datos, 654  
     escritor del registro, 655  
     gestor de bloques, 654  
     monitor de procesos, 655  
     punto de revisión, 655  
     servidor, 654  
 proceso de entrega de mensajes, 703  
 productividad, 513, 657, 742  
 producto cartesiano, 30, 36, 39  
 programas de aplicación, 11  
 Prolog, 137, 151, 162  
 propagación perezosa, 708  
 protocolos  
     basados en grafos, 792  
     basados en marcas temporales, 792  
     de acceso a directorios, 719  
     de acceso a directorios X.500, 720  
     de aplicaciones inalámbrico, 769  
     de bloqueo, 704  
     de compromiso, 698, 699  
         de dos fases, 665, 698, 699  
         de tres fases, 698, 701  
         en grupo, 789, 790  
     de control de concurrencia, 677  
     de mayoría, 705  
     de ordenación por marcas temporales, 540  
     de transferencia de hipertexto, 265  
     de validación, 792  
     globales-lectura, 798  
     globales-lectura-escritura/locales-lectura, 798  
     ligero de acceso a directorios, 719, 720  
     locales-lectura, 798  
     sesgado, 706  
     simple de acceso a objetos, 357, 748  
 proximidad, 634  
 proyección, 37  
     generalizada, 48  
     paralela, 684  
     temporal, 755  
 proyecto de directorio abierto, 645  
 pruebas de rendimiento, 741  
     familias de tareas, 741  
     OLAP, 742  
     OLTP, 742  
     OODB, 744  
     TPC, 742  
 PSM (Persistent Storage Module), 122  
 public, 111, 281  
 publicador, 906  
 puntero persistente, 318  
 punto de bloqueo, 534  
 punto fijo, 129, 159  
 PuntoFijo-Datalog, 159  
 puntos de almacenamiento, 591  
 puntos de revisión, 578, 580, 586  
     difuso, 580, 588  
 Python, 270  
 QBE, 27  
 QBE (Query-By-Example), 137, 144, 162  
 QMF (Query Management Facility), 165  
 quórum  
     de escritura, 706  
     de lectura, 706  
 R, árbol, véase índices, árbol R  
 réplica, 885  
     completa, 694  
     de actualización distribuida, 708  
     de datos, 694  
     maestro-esclavo, 707  
     multimaestro, 708  
 raíz, 332  
 RAID, 569, 767  
 RAID (Redundant Array of Independent Disks)  
     paridad con bits entrelazados, 378  
     paridad con bloques enrelazados, 379  
 RAID (Redundant Array of Independent Disks), 375  
 Ramos, Bill, 885  
 Rathakrishnan, Balaji, 885  
 Rdb de DEC, 23  
 Rdb de Oracle, 678  
 REA, 582  
 RealAudio, 766  
 realimentación de la relevancia, 634  
 reasignación de los sectores dañados, 371  
 recall, 640  
 rechazo falso, 640  
 recubrimiento canónico, 235, 235  
 recuperabilidad, 520, 770, 791  
     de elementos de datos de gran tamaño, 795  
 recuperación, 571, 579, 701  
     a un instante, 596  
     al reiniciar, 580, 587  
     basada en el registro histórico, 572  
     basada en la semejanza, 767  
     esquema, 567, 581  
     optimizaciones, 591  
     subsistema, 386  
 recuperación (recall), 640  
 recuperación de fallos, 18  
     algoritmo, 789  
 recuperación de información, 631  
     basada en la semejanza, 634  
 recursividad, 160  
     estructural, 348  
 recursos para presentaciones, 780  
 redes, 767  
     de área amplia, 666, 667  
     de área de almacenamiento, 371, 667  
     de área local, 666  
     de interconexión, 659  
 redundancia, 3, 375  
     P+Q, 380  
 referencia, 723  
 referencias cruzadas, 251  
 registro de escritura anticipada (REA), 582  
 registro histórico, 572, 796  
     con memoria intermedia, 581

- registro histórico (*cont.*)  
de operaciones, 795  
físico, 585  
registros, 585  
forzar, 582  
lógico, 585, 795  
registro de actualización, 572  
registro punto de revisión, 589  
registros de compensación, 586, 589  
registros, 572  
índice, 403  
reglas, 19, 151, 153, 615  
aumentatividad, 232  
de equivalencia, 477  
conjunto mínimo, 480  
de escritura de Thomas, 542  
de los cinco minutos, 736  
del minuto, 736  
descomposición, 233  
pseudotransitividad, 233  
reflexividad, 232  
transitividad, 232  
unión, 233  
regresión, 620  
lineal, 620  
reingeniería, 749  
relaciones, 13, 30, 173  
bitemporales, 754  
derivadas, 80  
desnormalizada, 737  
externa, 452  
identificadora, 189  
instantáneas, 755  
interna, 452  
referenciada, 35  
referenciente, 35  
temporales, 754  
uno a uno, 177  
uno a varios, 177  
varios a uno, 177  
varios a varios, 177  
relevancia, 633  
relevo, 768  
en caliente, 592  
reloj  
del sistema, 540  
lógico, 707  
rendimiento, 791  
de la reconstrucción, 380  
renombramiento, operación, 41, 68  
repetición de la historia, 587  
replicados, 664  
representación  
relacional  
agregación, 206  
atributos compuestos, 204  
combinación de esquemas, 204  
conjuntos de entidades débiles, 202  
conjuntos de entidades fuertes, 201  
conjuntos de relaciones, 202  
generalización, 205  
redundancia, 203  
repudio, 288  
requisitos funcionales  
especificación, 12, 170  
resolución del desbordamiento, 461  
respuesta a las preguntas, 643  
respuesta por desafío, 288  
restricciones, 238, 247  
de completitud, 194  
de consistencia, 3, 8  
de definición por condición, 195  
de integridad, 105  
referencial, 107  
legales, 225  
resultados, 783  
retículo, 193  
retroceso, 552  
en cascada, 534  
parcial, 552  
total, 552  
reunión, 45  
con bucles anidados en paralelo, 683  
con fragmentos y réplicas, 681, 682  
asimétricos, 681  
cruzada, 94  
de banda, 689  
de unión, 94  
en bucle anidado, 452  
indexada, 455  
por bloques, 454  
encauzada, 469  
espacial, 760  
externa, 51  
completa, 53, 93  
por la derecha, 52, 93  
por la izquierda, 52, 93  
minimización, 503  
natural, 45, 92  
operaciones, 92  
orden en profundidad por la izquierda, 491  
paralela  
algoritmos, 680  
por asociación, 459  
híbrida, 462  
por asociación dividida en paralelo, 683  
por división, 680, 680  
por mezcla, 455  
por ordenación-mezcla, 455  
temporal, 755  
zeta, 46  
revoke, 111, 282  
REXX, 863  
robustez, 710  
rol  
entidad, 173  
RosettaNet, 356  
RPC (Remote Procedure Call), 781  
transaccionales, 781  
Rys, Michael, 885  
saga, 794  
SAN (Storage Area Network), 667  
SATA (Serial ATA), 371, 373  
satisfce, 155, 225  
SAX (Simple API for XML), 350  
Schroeder, Bianca, 807  
SCSI (Small Computer System Interconnect), 371  
sectores, 370  
secuencialidad, 516  
de dos niveles, 797  
en cuanto a conflictos, 517  
asegurar, 533  
en cuanto a vistas, 517  
propiedad, 529  
protocolos  
basados en el bloqueo, 529  
basados en grafos, 537  
secuencias, 450  
secuencias de ejecución, 514  
segmento rectilíneo, 756  
seguridad, 157  
selección, 36, 347  
conjuntiva, 448  
de reemplazamiento, 473  
disyuntiva, 448  
paralela, 684  
temporal, 755  
select, 65, 66, 313, 887, 891  
selectividad, 484  
semántica  
programa, 155, 156  
reglas, 154  
semejanza del coseno, 634  
Sequel, 61  
Server-Side Javascript, 270  
servicio Web, 357  
servidores, 20  
de aplicaciones, 21  
de nombres, 696  
Web, 265, 779  
servlet, 267  
sesgo, 659, 681, 688  
de ejecución, 675, 680  
de los valores de los atributos, 676  
en una relación, 675  
SET (Secure Electronic Transaction), 788  
SGBD, 1  
SGML (Standard Generalized Markup Language), 329  
showplan, 887  
SIGMOD, conferencia de la ACM, 807  
sin compartimiento, 662  
sin conexión, 266  
sinónimos, 638  
sintaxis  
bidimensional, 144  
sistema  
dividido, 698  
sistema de archivos, 386  
sistema de bases de datos  
múltiples, 717  
sistema de posicionamiento global, 759  
sistema de procesamiento de archivos, 3  
sistema gestor de bases de datos, 1  
sistemas  
clientes, 653  
de archivos  
de diario, 374  
de colas, 734

- sistemas (*cont.*)  
 de discos compartidos, 667  
 de diseño asistido por computadora, 757  
 de gestión de flujos de trabajo, 665  
 de información geográfica, 756  
 de planificación de los recursos de las empresas, 785  
 de tiempo real, 790  
 heredados, 748  
 monousuario, 652  
 multiusuario, 652  
 paralelos, 657  
 persistentes de C++, 319  
 relacionales, 323  
 remotos de copia de seguridad, 595  
 servidores, 653  
   de consultas, 654  
   de datos, 654, 656  
   de transacciones, 654  
 sistemas de bases de datos  
   cliente-servidor, 651  
   distribuidos, 651, 663, 693  
   distribuidos y heterogéneos, 664  
   múltiples, 664, 796  
   paralelos, 651, 673  
 sitio, 663  
   principal, 591, 705  
   remoto de copia de seguridad, 591  
   secundario, 591  
 smallint, 102  
 SOAP, 357  
 SOAP (Simple Object Access Protocol), 748  
 sobrecarga, 319  
 software envolvente, 358  
 Solaris, 859  
 soporte, 621  
 span, 755  
 SQL, 61, 323, 507  
   dinámico, 62  
 dominios  
   char, 63  
   double precision, 63  
   float, 63  
   int, 63  
   numeric, 63  
   real, 63  
   smallint, 63  
   varchar, 63  
   incorporado, 62, 112  
 SQL Server de Microsoft, 59  
 SQL-92, 62  
 Standard Generalized Markup Language, 329  
 Storage Area Network, 371  
 subárboles disjuntos, 538  
 subastas, 787  
   inversas, 787  
 subclase, 192  
 subconsultas, 78  
   escalares, 130  
 subesquemas, 6  
 subexpresiones comunes, 503  
 subtabla, 308  
 subtareas, 791  
 sufijo, 722  
 sumas de comprobación, 371  
 Sun Microsystems, 321, 745  
 superclase, 192  
 superclase-subclase, 191  
 superclave, 34, 178, 225  
 superusuario, 279  
 supuesto de fallo-parada, 567  
 suscriptores, 907  
 sustitución de bloques utilizados menos recientemente, 385  
 Sybase, 59  
 synset, 639  
 System R, 23, 59, 61, 474, 491, 527, 597, 828, 859  
 término, 632  
 tabla de páginas desfasadas, 588  
 TablaPáginasDesfasadas, 589  
 tablas, 29, 30  
   de dimensiones, 614  
   de hechos, 614  
   dinámica, 603  
 tabulación cruzada, 603  
 tag library, 271  
 tarea, 781  
 tarjetas inteligentes, 288  
 Tcl, 808  
 teléfonos móviles, 184, 768  
 teoría de colas, 735  
 terabyte, 369  
 Teradata, 687  
 Term Frequency (TF), 633  
 terminales, 767  
 test  
   de potencia, 744  
   de productividad, 744  
 texto completo, 632  
 TF (Term Frequency), 633  
 Thomas, *véase* reglas, de escritura de Thomas  
 tiempo  
   de acceso, 372  
   de búsqueda, 372  
     medio, 372  
   de compromiso, 592  
   de latencia medio, 372  
   de latencia rotacional, 372  
   de recuperación, 592  
   de respuesta, 657  
   de servicio, 741  
   de transacción, 754  
   en SQL, 754  
   límite  
     estricto, 790  
     firme, 790  
     flexible, 790  
   medio  
     de reparación, 376  
     de respuesta, 514  
     entre fallos, 372  
     entre pérdidas de datos, 376  
   para concluir, 742  
   válido, 754  
 tiempo límite, 790  
 tipos de datos  
   coerción, 102  
   compatibles, 102  
   de conjunto, 318  
   de fila, 305  
   definidos por el usuario, 304  
   estructurados, 102  
   más concreto, 307  
 tipos de dominios, 63, 103  
 Tk, 808  
 tolerancia ante fallos, 662  
 Top End, 777  
 TP, 777  
 TPS (transacciones por segundo), 743  
 transacciones, 90, 777  
   abortada, 510  
   activa, 510  
   anidada, 794  
   cancelar, 511  
   compensadora, 510, 794  
   comprometida, 510  
   concurrentes, 579  
   de actualización, 548  
   de corta duración, 792  
   de sólo lectura, 548  
   definición, 18, 507, 507  
   distribuida, 697  
   dudosas, 701  
   ejecución  
     concurrente, 513  
     secuencial, 513  
   electrónica segura, 788  
   fallida, 510  
   fallo, 567  
   flujo de tareas, 781  
   flujo de trabajo, 781  
   global, 663, 697, 796  
   instantánea consistente, 707  
   interactiva  
     compleja, 795  
     de larga duración, 799  
   larga duración, 791  
   llamada a procedimientos remotos, 653  
   local, 663, 697, 796  
   múltiples, 513  
   modificación  
     diferida, 573  
     inmediata, 575  
   multinivel, 794  
   parcialmente comprometida, 510  
   procesadas por minilotes, 741  
   reiniciar, 511  
   retrocedida, 510  
   retroceso, 579, 585  
     en cascada, 521  
     terminada, 510  
 Transaction Processing Performance Council, 742  
 transferencia  
   de prestigio, 636  
   del control, 592  
 transformación, 614  
 transformar, 613

- transparencia
  - de la fragmentación, 696
  - de la réplica, 696
  - de la ubicación, 696
  - de los datos, 696
- traza de auditoría, 285
- triangulación, 756
- tries, 440
- triviales, 226
- tupla, 30
- Turing
  - máquina universal, 11
  - premio, 23
- Tuxedo, 777
- UDDI, 357
- Ultrium, 383
- UML (Unified Modeling Language), 210
- union, 71
- unión, 38, 71, 464
- union join, 94
- unique, 106
- Universel Temps Coordoné, 755
- Unix, 807, 859
- update, 111, 887
- upsert, 834
- URL (Uniform Resource Locator), 263
- USB (Universal Serial Bus), 368, 371
- UTC, 755
- utilización, 513
- valores
  - anterior, 572
  - continuos, 618
  - nuevo, 572
- variables
  - de correlación, 493
  - externas, 783
  - tupla, 30, 68
- varrays, 835
- VBScript, 270, 747
- vector de división, 674
  - por rangos equilibrado, 676
- vector de versiones, 771
- velocidad de transferencia de datos, 372
- ventana, 611
- versiones, 547
  - multiconjunto, 70
- vídeo
  - servidor, 767
- Virtual Private Database, 284
- vistas, 81, 82, 155, 283
  - actualizables, 89
  - definición, 61, 82
  - materializadas, 83, 494, 503
    - mantenimiento, 83, 495, 613
    - mantenimiento diferido, 495, 738
    - mantenimiento incremental, 495
    - mantenimiento inmediato, 495, 738
    - selección, 498
  - no recursiva, 154
  - paramétricas, 122
  - recursiva, 83, 154
- Visual Basic, 115, 779, 905
- Visual C++, 260
- visualización de datos, 624
- VM, 859
- volcado de archivo, 584
  - difuso, 585
  - volcar, 584
- VPD, 284
- VRML (Virtual Reality Markup Language), 265
- Waas, Florian, 885
- WAN, 667
- WAP (Wireless Application Protocol), 769
- Web, 262
- WebSphere, 270
- where, 66, 84
- Wide Area Networks, 667
- Windows, 859
  - 2000, 859
  - XP, 859
- with, 80, 94, 134
- WML (Wireless Markup Language), 769
- World Wide Web, 262
- WSDL, Web Services Descripción Language, 357
- XML (Extensible Markup Language), 329
  - datos relacionales publicados, 353
- XPath, 340, 341
- XPS de Informix, 687
- XQJ, 347
- XQuery, 340, 343
- XSLT (XSL Transformations), 340
- XSLT (XSLT Transformations), 347, 347
  - keys, 348
- Yahoo, 623, 645
- z/OS, 859
- Zope, 270
- Zwilling, Michael, 885

## ACERCA DEL LIBRO

La quinta edición del libro *Fundamentos de bases de datos* de los autores Silberschatz, Korth y Sudarshan ofrece todo lo necesario para una profunda comprensión de los sistemas de bases de datos. Los autores explican conceptos fundamentales de la gestión de bases de datos con todo el detalle y atención que los lectores de este clásico libro de texto esperan. Se examinan en profundidad los lenguajes de consultas, el diseño de esquemas, el desarrollo de aplicaciones, la implementación de sistemas, el análisis de datos y las arquitecturas de las bases de datos. El libro es apropiado tanto para cursos introductorios como avanzados, así como referencia para profesionales.

La quinta edición de *Fundamentos de bases de datos* conserva el estilo general de las ediciones anteriores a la vez que evoluciona su contenido y organización para reflejar los cambios que se han producido al diseñar, gestionar y utilizar las bases de datos. Algunos de los contenidos de esta nueva edición son:

- Nueva organización de los capítulos, adelantando el estudio de SQL.
- Tratamiento ampliado de SQL, incluyendo SQL:2003.
- Una nueva parte dedicada al diseño de bases de datos con un tratamiento ampliado de la normalización y los datos temporales.
- Tratamiento ampliado y actualizado de XML.
- Tratamiento detallado de la implementación y las arquitecturas de los sistemas de bases de datos.
- Tratamiento ampliado del diseño y desarrollo de aplicaciones.
- Capítulos sobre minería y análisis de datos, y de recuperación de información.
- Estudios de casos que incluyen las últimas versiones de IBM DB2, Oracle, Microsoft SQL Server y PostgreSQL (nuevo en esta edición).
- Énfasis en los aspectos prácticos, aplicaciones e implementación, acompañados de un tratamiento intuitivo de los conceptos teóricos clave.

## ACERCA DE LOS AUTORES

**Abraham Silberschatz** (Dr. por la Universidad estatal de Nueva York en Stony Brook) es catedrático de Informática en la Universidad de Yale. Antes de trabajar para Yale, fue vicepresidente del Information Sciences Research Center en Bell Laboratories, Murray Hill, Nueva Jersey. Anteriormente fue catedrático del Departamento de Informática en la Universidad de Texas en Austin. Sus intereses en investigación incluyen los sistemas operativos, los sistemas de bases de datos, los sistemas en tiempo real, los sistemas de almacenamiento, la administración de redes y los sistemas distribuidos. El profesor Silberschatz es miembro de las sociedades ACM e IEEE. Ha recibido el premio Taylor L. Booth Education Award de la IEEE en el 2000, el premio Karl V. Karlstrom Outstanding Educator Award de la ACM en 1998, el premio SIGMOD Contribution Award de la ACM en 1997 y el premio Computer Society Outstanding Paper Award de la IEEE. También es autor del libro *Fundamentos de sistemas operativos*, publicado en castellano por McGraw-Hill.

**Henry F. Korth** (Dr. por la Universidad de Princeton) es catedrático Weiseman y director del departamento de Ciencias e Ingeniería Informática en la Universidad de Lehigh. Antes de trabajar para Lehigh, fue director de investigación en principios de bases de datos de Bell Laboratories, Murray Hill, Nueva Jersey. Entre sus líneas de investigación se encuentran la gestión de datos XML, datos en Web, sistemas de bases de datos en memoria principal, sistemas en tiempo real, sistemas paralelos y otros temas. Antes de trabajar en Bell Laboratories, el profesor Korth fue vicepresidente de Panasonic Technologies y director del Matsushita Information Technology Laboratory. Anteriormente ha sido catedrático asociado en el departamento de Informática en la Universidad de Texas en Austin e investigador en el T. J. Watson Research Center de IBM. El profesor Korth es miembro de las sociedades ACM e IEEE.

**S. Sudarshan** (Dr. por la Universidad de Wisconsin, Madison) es catedrático en el Departamento de Ciencias e Ingeniería Informática del Indian Institute of Technology (IIT) en Bombay. Anteriormente ha sido miembro del equipo técnico del grupo de investigación de bases de datos de Bell Laboratories, Murray Hill, Nueva Jersey. El profesor Sudarshan es autor de alrededor de 60 artículos en diferentes áreas de los sistemas de bases de datos y tiene 13 patentes. Entre las líneas de investigación del profesor Sudarshan se encuentran el procesamiento y optimización de consultas, la autorización y la consulta bajo palabras clave en bases de datos. Además de ser arquitecto de varios sistemas software relacionados con la implementación de los sistemas de bases de datos, también ha sido responsable de construir y mantener varias aplicaciones de bases de datos utilizadas en el IIT de Bombay.



9 788448 146443

ISBN: 84-481-4644-1