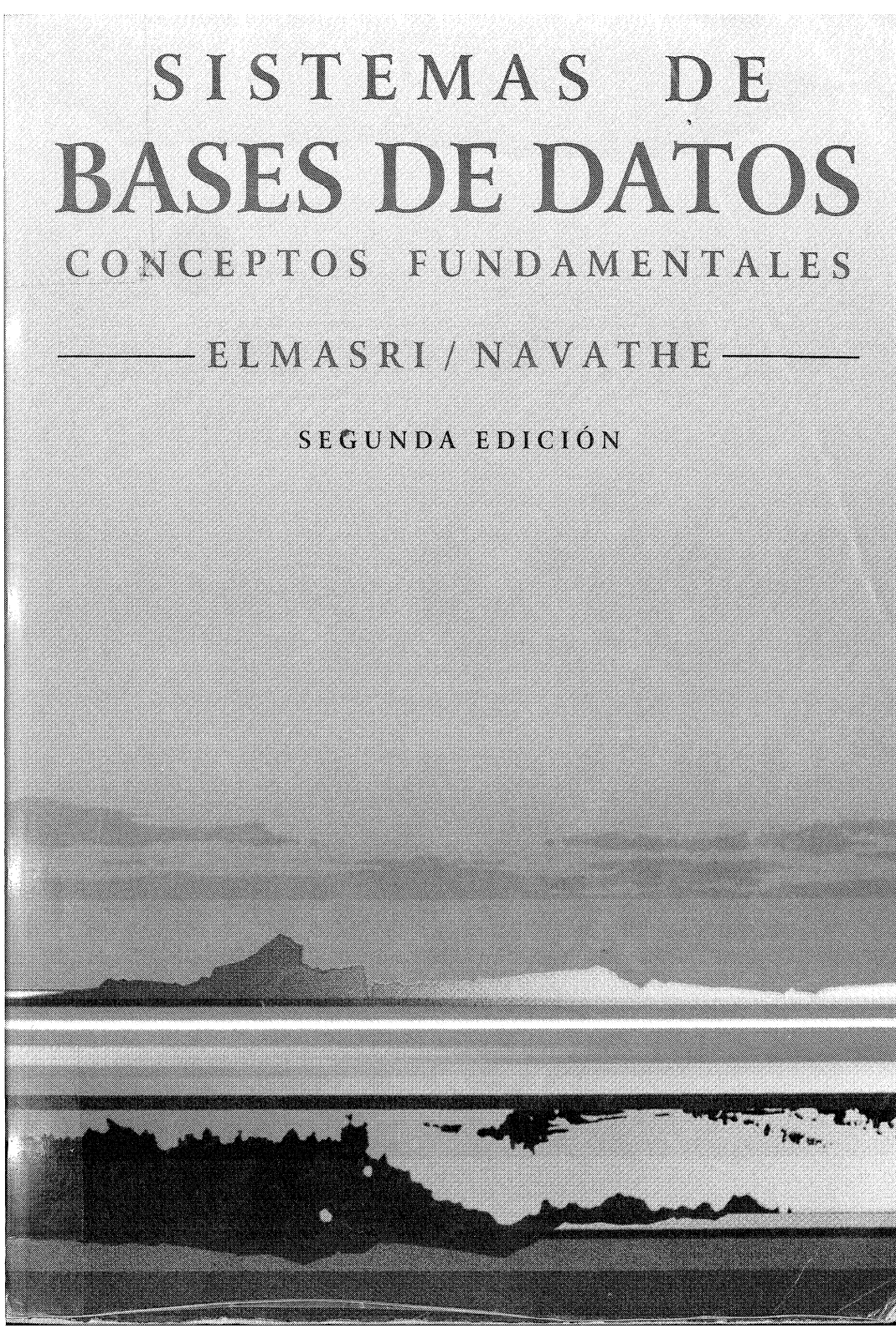
SISTEMAS DE BASES DE DATOS

CONCEPTOS FUNDAMENTALES

ELMASRI / NAVATHE

SEGUNDA EDICIÓN



SISTEMAS DE BASES DE DATOS

Conceptos fundamentales

SEGUNDA EDICIÓN

Universidad Abierta Interamericana
RECTORADO

Ramez Elmasri
Department of Computer Science Engineering
University of Texas, Arlington

Shamkant B. Navathe
College of Computing
Georgia Institute of Technology

Versión en español de

Roberto Escalona García
México, D.F.

093 028583

Con la colaboración de

Felipe López Gamino
Instituto Tecnológico Autónomo de México

y

Arantza Illarramendi
Universidad del País Vasco
San Sebastián, España

Addison-Wesley Iberoamericana
Argentina • Chile • Colombia • España • Estados Unidos
México • Perú • Puerto Rico • Venezuela

PREFACIO

Este libro presenta los conceptos fundamentales requeridos para diseñar, utilizar e implementar sistemas de bases de datos. Nuestra presentación hace hincapié en los fundamentos del modelado y diseño de bases de datos, en los lenguajes y recursos que ofrecen los sistemas de gestión de bases de datos y en las técnicas de implementación de sistemas. Esta obra puede usarse como libro de texto para un curso sobre sistemas de bases de datos con duración de uno o dos semestres en los últimos años de licenciatura o en estudios de posgrado, y también como libro de consulta. Suponemos que los lectores cuentan con conocimientos elementales sobre programación y estructuras de datos, y que han tenido cierta experiencia en la organización básica de los computadores.

Decidimos comenzar la PARTE I con una exposición de conceptos de ambos extremos del espectro de las bases de datos: principios del modelado conceptual y técnicas del almacenamiento físico de archivos. Concluimos el libro en la PARTE VI con un análisis de lo más influyente entre los modelos nuevos de bases de datos, como el modelo orientado a objetos y el modelo deductivo, junto con un panorama general de las tendencias que están surgiendo en la tecnología de bases de datos. Entre estos dos puntos —desde la PARTE II hasta la V— se ofrece al lector un tratamiento a fondo de los aspectos más importantes en torno a los fundamentos de las bases de datos.

Entre las características clave de la segunda edición podemos mencionar las siguientes:

- Una organización independiente y flexible, diseñada para adaptarse a necesidades individuales.
- Cobertura completa del modelo relacional, y un tratamiento actualizado de SQL2 en la PARTE II.
- Sinopsis de los sistemas de legado —de red y jerárquico— en la PARTE III.
- Un capítulo nuevo, muy completo, que presenta las bases de datos orientadas a objetos, y otro que trata las bases de datos deductivas.

- Un ejemplo desarrollado a lo largo del libro, COMPAÑÍA, que permite al lector comparar diferentes enfoques para llevar a la práctica una misma aplicación.
- Cobertura del diseño de bases de datos, incluidos el diseño conceptual, las técnicas de normalización y el diseño físico.
- Presentaciones actualizadas sobre los conceptos de implementación de sistemas de gestión de bases de datos (SGBD), entre ellos el procesamiento de consultas, el control de concurrencia, la recuperación y la seguridad.
- Modernísima cobertura de los avances más recientes, incluidos panoramas de las bases de datos distribuidas, activas, temporales y de multimedia.

Contenido del libro

La PARTE I describe los conceptos básicos indispensables para entender el diseño y la implementación de bases de datos. Los primeros dos capítulos son una introducción a las bases de datos, a los usuarios representativos, a los conceptos de SGBD y a la arquitectura de estos sistemas. En el capítulo 3 se estudian los conceptos del modelo de entidad-vínculo (ER), aprovechándolos para ilustrar el diseño conceptual de bases de datos. En el capítulo 4 se describen los métodos básicos de organización de archivos de registros en disco, incluyendo los de dispersión estática y dinámica. En el capítulo 5 se describen las técnicas de indización de archivos, entre ellas las de árboles B y B^+.

La PARTE II se ocupa del modelo de datos relacional. El capítulo 6 describe el modelo relacional básico, sus restricciones de integridad y operaciones de actualización, así como las operaciones del álgebra relacional. Además cuenta con una sección opcional en la que se describe el diseño de esquemas relacionales a partir de diagramas ER conceptuales. El capítulo 7 ofrece un panorama detallado del lenguaje SQL —actualizado en la segunda edición para cubrir características de la norma SQL2— El capítulo 8 presenta los lenguajes formales del cálculo relacional, e incluye sinopsis de los lenguajes QUEL y QBE. En el capítulo 9 se analizan sistemas comerciales de bases de datos relacionales, y se hace un resumen detallado del sistema DB2 de IBM.

En la PARTE III se estudian los llamados sistemas de bases de datos de legado, a saber, los sistemas de red y jerárquico, sobre los que se han construido muchas aplicaciones comerciales de bases de datos, en particular las que utilizan bases de datos de gran tamaño y sistemas de procesamiento de transacciones. Los modelos de red y jerárquico se tratan en los capítulos 10 y 11, respectivamente. Primero se describen los dos modelos sin relacionarlos con SGBD específicos; una sección opcional en cada capítulo explica cómo convertir el diseño conceptual de un esquema de base de datos realizado en el modelo ER en un esquema de red o jerárquico. Además, ambos capítulos contienen un panorama general de un sistema comercial: IDMS para el modelo de red e IMS para el jerárquico.

La PARTE IV cubre varios temas relacionados con el diseño de bases de datos. En primer lugar, en los capítulos 12 y 13, explicamos los formalismos, la teoría y los algoritmos que se han desarrollado para diseñar, mediante normalización, bases de datos relacionales. En este material entran las dependencias funcionales y de otros tipos, y las formas normales de las relaciones. En el capítulo 12 se presenta, en términos intuitivos, la normalización paso por paso, y los algoritmos de diseño relacional se explican en el capítulo 13, en el cual se definen

también otros tipos de dependencias, como las multivaluadas y las de reunión. En el capítulo 14 se presenta un panorama general de las diferentes fases que constituyen el proceso de diseño de bases de datos para aplicaciones de mediano y gran tamaño, además de que se analizan cuestiones del diseño físico de bases de datos aplicables a los SGBD relacionales, de red y jerárquicos.

La PARTE V se ocupa de las técnicas empleadas para implementar sistemas de gestión de bases de datos (SGBD). En el capítulo 15 se describe la implementación del catálogo de los SGBD, componente vital de todo SGBD. El capítulo 16 presenta las técnicas para procesar y optimizar consultas especificadas en un lenguaje de base de datos de alto nivel, como SQL, y analiza varios algoritmos para implementar operaciones de bases de datos relacionales. En los capítulos 17 al 19 se analizan técnicas de procesamiento de transacciones, control de concurrencia y recuperación; todo este material se ha enmendado para esta segunda edición. En el capítulo 20 se estudian técnicas de seguridad y de autorización para las bases de datos.

La PARTE VI cubre varios temas avanzados. En el capítulo 21 se analizan conceptos de abstracción de datos y de modelado semántico de datos, y se extiende el modelo ER para incorporar estas ideas, produciendo el modelo de datos ER extendido (EER). Entre los conceptos abordados están las subclases, la especialización, la generalización y las categorías. También se estudian las restricciones de integridad y el diseño conceptual de transacciones, y se hace una sinopsis de los modelos de datos funcional, relacional anidado, estructural y semántico. El capítulo 22 ofrece una introducción muy completa a las bases de datos orientadas a objetos, y menciona ejemplos de dos sistemas comerciales. En el capítulo 23 se estudian las bases de datos distribuidas y la arquitectura cliente-servidor. Con la aparición de estaciones de trabajo potentes y redes de comunicación de alta velocidad, se han hecho factibles las bases de datos verdaderamente distribuidas. El capítulo 24 es una introducción a los conceptos de sistemas de bases de datos deductivas. Por último, en el capítulo 25 se examinan las tendencias de la tecnología de bases de datos y se hace un análisis de varias tecnologías y aplicaciones en esta área que están surgiendo en la actualidad. Entre las tecnologías de la siguiente generación se encuentran las bases de datos activas, temporales, espaciales, científicas y de multimedia. Entre las aplicaciones más recientes se cuentan el diseño y la fabricación en el área de ingeniería, los sistemas de oficina y de apoyo a la toma de decisiones, así como aplicaciones biológicas.

En el apéndice A se encontrarán diversas notaciones diagramáticas alternativas para representar esquemas conceptuales ER, y el profesor puede utilizarlas, si así lo desea, en vez de la de los autores. El apéndice B contiene ciertos parámetros físicos importantes de los discos, y el apéndice C hace una breve comparación de los diversos modelos de datos analizados en el libro.

Pautas para utilizar este libro

Son muchas las formas de impartir un curso sobre esta área. Los capítulos de las partes I a III, en el orden en que se presentan, pueden constituir un curso de introducción a los sistemas de base de datos, aunque el profesor puede alterar el orden si lo prefiere. Es posible omitir algunos capítulos y secciones, y el profesor puede agregar otros capítulos del resto del libro, según la orientación de su curso. Si se hace hincapié en las técnicas de implementación de sistemas, se pueden utilizar capítulos selectos de la PARTE V. Si la orientación es hacia

el diseño de bases de datos, pueden incluirse capítulos de la PARTE IV. Las secciones marcadas con una estrella (★) podrían omitirse si se desea un tratamiento menos detallado del tema abordado en el capítulo correspondiente.

El capítulo 3, sobre el modelado conceptual mediante el modelo de entidad-vínculo (ER), ofrece una importante visión conceptual de los datos. Sin embargo, es posible omitirlo o tratarlo más adelante si el profesor así lo desea. Si los estudiantes ya han tomado un curso sobre técnicas de organización de archivos, se les puede pedir que lean partes de los capítulos 4 y 5 para repasar los conceptos de este tema. El libro está escrito de modo que los temas puedan estudiarse en diversos órdenes, y en el diagrama que se presenta a continuación se muestran las principales dependencias entre los capítulos. Como puede verse, es posible comenzar con cualquiera de los modelos de datos (relacional, ER, de red, jerárquico) después de los capítulos introductorios. También es factible tratar la organización de archivos y la indización al principio del curso o posponer estos temas. Si se deja para después el tratamiento del modelo ER, las secciones sobre la transformación del ER de los capítulos 6, 10 y 11 pueden estudiarse una vez que se haya examinado el material del capítulo 3.

En el caso de un curso de un solo semestre basado en este libro, se pueden dejar algunos capítulos como lecturas adicionales. Los capítulos 4, 5, 14, 15 y 25 podrían ser adecuados para este propósito. El libro también puede servir para una secuencia de dos semestres. El primer curso, "Introducción a los sistemas de bases de datos", a nivel de segundo, tercero o cuarto año de licenciatura, podría cubrir la mayor parte del contenido de

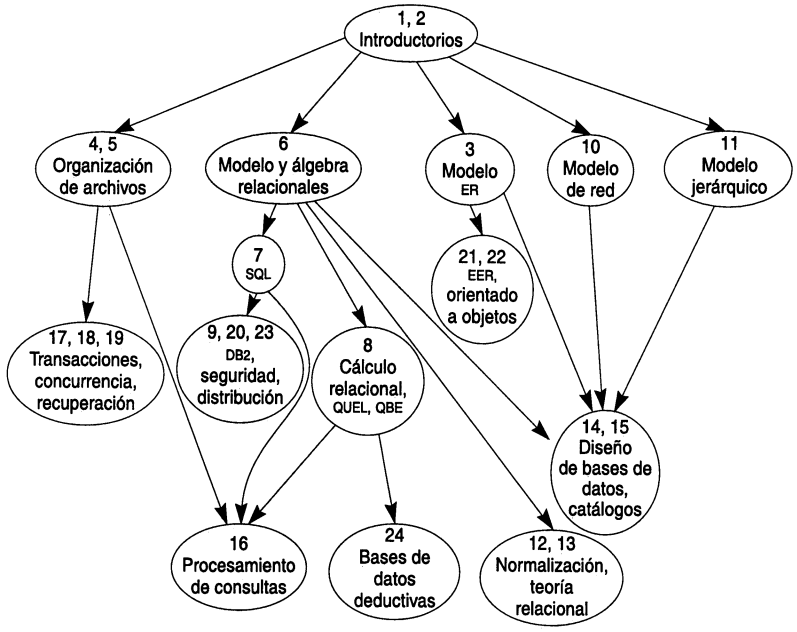


Diagrama de dependencias

los capítulos 1 a 12. El segundo curso, "Técnicas de diseño e implementación de bases de datos", a nivel del último año de licenciatura o del primer año de posgrado, puede cubrir los capítulos restantes, además de aquellos que se hayan omitido en el primer curso. La PARTE VI puede servir como material introductorio para temas adicionales que desee contemplar el profesor. Es posible usar selectivamente capítulos de la PARTE VI en cualquiera de los dos semestres, y estudiar, además del contenido del libro, material que describa el SGBD al que puedan tener acceso los estudiantes en la institución local.

Existen varios suplementos para la versión en inglés del libro, de los cuales el principal es una guía para el profesor, que abarca soluciones de la mayor parte de los ejercicios del libro, así como un análisis de posibles enfoques para impartir el material de cada capítulo. Suplementos adicionales están disponibles por ftp anónimo desde el sitio bc.aw.com dentro del directorio bc/elmasri.

Agradecimientos

Es para nosotros un placer hacer un reconocimiento a la ayuda y las contribuciones que para este proyecto ofrecieron muchas personas. Nuestro editor, Dan Joraanstad, nos animó y motivó constantemente para terminar la revisión del libro. Quisiéramos agradecer las aportaciones brindadas por quienes hicieron un ejercicio crítico de algunas partes de la segunda edición y sugirieron mejoras a la primera. Entre ellos están Rafi Ahmed, Antonio Albano, David Beech, Jose Blakeley, Panos Chrysanthis, Suzanne Dietrich, Vic Ghorpadey, Goetz Graefe, Eric Hanson, Junguk L. Kim, Roger King, Vram Kouramajian, Vijay Kumar, John Lowther, Sanjay Manchanda, Toshimi Minoura, Inderpal Mumick, Ed Omiecinski, Girish Pathak, Raghu Ramakrishnan, Ed Robertson, Eugene Sheng, David Stotts, Marianne Winslett y Stan Zdonick. También nos gustaría expresar nuestro agradecimiento a los estudiantes de la University of Texas en Arlington y del Georgia Institute of Technology, por haber utilizado borradores del material nuevo de la segunda edición y leído con cuidado los manuscritos.

Queremos reiterar nuestro reconocimiento a quienes nos presentaron su crítica de la primera edición de este libro e hicieron contribuciones a ella. Entre ellos debemos mencionar a Alan Apt, Don Batory, Scott Downing, Dennis Heimbigner, Julia Hodges, Yannis Ioannidis, Jim Larson, Dennis McLeod, Per-Ake Larson, Rahul Patel, Nicholas Roussopoulos, David Stemple, Michael Stonebraker, Frank Tompa y Kyu-Young Whang. Muchos estudiantes de posgrado de la University of Florida nos proporcionaron valiosas sugerencias para la primera edición.

A Sham Navathe le gustaría agradecer la ayuda secretarial de Sharon Grant en la primera edición, y a Gwen Baker y Jalisa Norton en la segunda.

Por último, reconocemos con gratitud el apoyo, el aliento y la paciencia de nuestras familias.

R.E.
S.B.N.

ACERCA DE LOS AUTORES

Ramez A. Elmasri es profesor asociado de ciencias de la computación en la University of Texas en Arlington. Sus temas de investigación preferidos son las bases de datos orientadas a objetos y los sistemas distribuidos, el modelado de datos y los lenguajes de consulta, así como las bases de datos temporales. Muy conocido por sus investigaciones para extender el modelo de entidad-vínculo, los trabajos actuales del profesor Elmasri se orientan a la incorporación del tiempo en los sistemas de bases de datos. En la década de 1980 fue uno de los principales investigadores del Computer Sciences Center de Honeywell en Minnesota, y trabajó en el diseño e implementación de un sistema prototipo de gestión de bases de datos distribuidas: DDTS. Es uno de los autores de *Temporal Databases: Theory, Design and Implementation,* y ha publicado más de 40 artículos sobre teoría y diseño de bases de datos.

Shamkant B. Navathe es profesor de computación en el Georgia Institute of Technology. Entre sus aportaciones a la investigación caben el modelado y la conversión de bases de datos, el diseño de bases de datos lógicas, el diseño de bases de datos distribuidas y la integración de bases de datos. Ha sido asesor para importantes distribuidores de computadores, como Honeywell, Siemens y DEC. El profesor Navathe es editor asociado de *Computing Surveys*, que publica la Association for Computing Machinery, y es el editor de la serie sobre sistemas y aplicaciones de bases de datos de la Benjamin/Cummings. Ha publicado un gran número de artículos, y además es coautor de *Conceptual Database Design: An Entity-Relationship Approach.*†

†Existe versión en español de esta obra, publicada por Addison-Wesley Iberoamericana, 1994. (*N. del E.*)

ÍNDICE GENERAL RESUMIDO

ÍNDICE GENERAL

PARTE III MODELOS DE DATOS Y SISTEMAS CONVENCIONALES

PARTE IV DISEÑO DE BASES DE DATOS

PARTE V TÉCNICAS DE IMPLEMENTACIÓN DE SISTEMAS

CAPÍTULO 1

Bases de datos y sus usuarios

En la sección 1.1 de este capítulo comenzaremos por definir qué es una base de datos, y luego definiremos otros términos fundamentales. En la sección 1.2 presentaremos un ejemplo sencillo de una base de datos, UNIVERSIDAD, a fin de ilustrar nuestro análisis. En la sección 1.3 describiremos algunas de las principales características de los sistemas de bases de datos, y en las secciones 1.4 y 1.5 clasificaremos los distintos tipos de personas que trabajan con sistemas de bases de datos e interactúan con ellos. Las secciones 1.6, 1.7 y 1.8 son un estudio más a fondo de las diversas capacidades de los sistemas de bases de datos y de las implicaciones que presenta la adopción del enfoque de bases de datos. Por último, el capítulo se resume en la sección 1.9.

El lector que tan sólo desee una breve introducción a los sistemas de bases de datos puede pasar por alto o examinar superficialmente las secciones 1.6 a 1.8, y continuar después con el capítulo 2.

1.1 Introducción

Las bases de datos y su tecnología están teniendo un impacto decisivo sobre el creciente uso de los computadores. No es exagerado decir que las bases de datos desempeñarán un papel crucial en casi todas las áreas de aplicación de los computadores, como los negocios, la ingeniería, la medicina, el derecho, la educación y la biblioteconomía, por mencionar sólo unas cuantas. El término *base de datos* es tan común que debemos comenzar por definir qué quiere decir. Nuestra definición inicial ha de ser bastante general.

Una **base de datos** es un conjunto de datos relacionados entre sí. Por **datos** entendemos hechos conocidos que pueden registrarse y que tienen un significado implícito. Por ejemplo, consideremos los nombres, números telefónicos y direcciones de personas que conocemos. Tal vez hayamos registrado estos datos en una libreta de direcciones indizada, o quizá lo hayamos hecho en un disquete, empleando un computador personal y software del tipo de DBASE IV o V, PARADOX o EXCEL. Se trata de un conjunto de datos relacionados entre sí y que tienen un significado implícito; por tanto, constituyen una base de datos.

La definición anterior es muy general; por ejemplo, podemos considerar el conjunto de palabras que forman esta página de texto como datos relacionados entre sí, de modo que son una base de datos. Pero la acepción común del término *base de datos* suele ser más restringida. Una base de datos tiene las siguientes propiedades implícitas:

- Una base de datos representa algún aspecto del mundo real, en ocasiones llamado **minimundo** o **universo de discurso**. Las modificaciones del minimundo se reflejan en la base de datos.
- Una base de datos es un conjunto de datos lógicamente coherente, con cierto significado inherente. Una colección aleatoria de datos no puede considerarse propiamente una base de datos.
- Toda base de datos se diseña, construye y puebla con datos para un propósito específico. Está dirigida a un grupo de usuarios y tiene ciertas aplicaciones preconcebidas que interesan a dichos usuarios.

En otras palabras, una base de datos tiene una fuente de la cual se derivan los datos, cierto grado de interacción con los acontecimientos del mundo real y un público que está activamente interesado en el contenido de la base de datos.

Las bases de datos pueden ser de cualquier tamaño y tener diversos grados de complejidad. Por ejemplo, la lista de nombres y direcciones antes mencionada puede contener apenas unas cuantas centenas de registros, cada uno de ellos con una estructura muy simple. Por otro lado, el catálogo de una biblioteca grande puede contener medio millón de tarjetas clasificadas por categorías distintas —apellido del primer autor, tema, título, etc.— y ordenadas alfabéticamente en cada categoría. Las autoridades fiscales mantienen una base de datos todavía más grande y compleja para llevar el control de las declaraciones fiscales que presentan los contribuyentes. Si suponemos que en los Estados Unidos hay 100 millones de contribuyentes y que cada uno presenta en promedio cinco formas, con aproximadamente unos 200 caracteres de información por formulario de declaración, las autoridades fiscales de ese país manejarían una base de datos con $100*(10^6)*200*5$ de caracteres (bytes) de información. Suponiendo que dichas autoridades conservan las últimas tres declaraciones de cada contribuyente, además de la actual, tendríamos una base de datos de $4*(10^{11})$ bytes. Esta enorme cantidad de información debe organizarse y controlarse para que los usuarios puedan buscar, obtener y actualizar los datos cuando sea necesario.

La generación y el mantenimiento de las bases de datos pueden ser manuales o mecánicos. El catálogo en tarjetas de una biblioteca es un ejemplo de base de datos que se puede crear y mantener manualmente. Las bases de datos computarizadas se pueden crear y mantener con un grupo de programas de aplicación escritos específicamente para esa tarea, o bien mediante un sistema de gestión de bases de datos.

Un **sistema de gestión de bases de datos** (SGBD; en inglés, *database management system:* DBMS) es un conjunto de programas que permite a los usuarios crear y mantener una base de datos. Por tanto, el SGBD es un sistema de software *de propósito general* que facilita

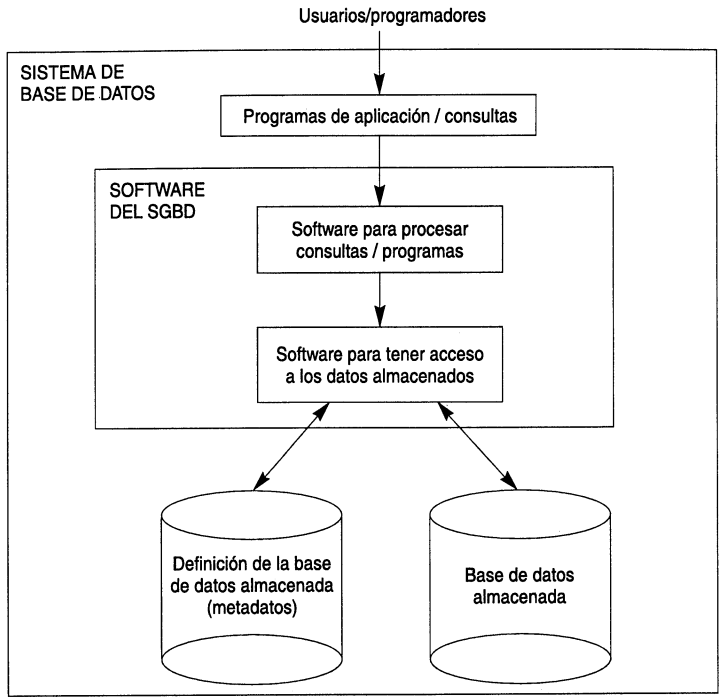


Figura 1.1 Entorno simplificado de un sistema de base de datos.

el proceso de definir, construir y manipular bases de datos para diversas aplicaciones. Para **definir** una base de datos hay que especificar los tipos de datos, las estructuras y las restricciones de los datos que se almacenarán en ella. **Construir** una base de datos es el proceso de guardar los datos mismos en algún medio de almacenamiento controlado por el SGBD. En la **manipulación** de una base de datos intervienen funciones como consultar la base de datos para obtener datos específicos, actualizar la base de datos para reflejar cambios en el minimundo y generar informes a partir de los datos.

No hace falta un software de SGBD de propósito general para implementar una base de datos computarizada. Podríamos escribir nuestro propio conjunto de programas para crear y mantener la base de datos, con lo cual estaríamos creando de hecho nuestro propio software de SGBD *de propósito específico*. En todo caso, ya sea que utilicemos un SGBD de propósito general o no, casi siempre requeriremos un software de gran capacidad para manipular la base de datos, además de la base de datos misma. Al conjunto formado por la base de datos y el software lo llamaremos **sistema de base de datos**. La figura 1.1 ilustra estas ideas.

1.2 Un ejemplo

Consideremos un ejemplo que casi todos los lectores conocerán: una base de datos UNIVERSIDAD para mantener información acerca de los estudiantes, cursos y notas en un entorno

universitario. La figura 1.2 ilustra la estructura de la base de datos y algunos datos de muestra para ella. La base de datos está organizada en cinco archivos, cada uno de los cuales almacena registros de datos del mismo tipo.† El archivo ESTUDIANTE contiene datos de todos los estudiantes; el archivo CURSO contiene datos de todos los cursos; el archivo SECCIÓN contiene datos de todas las secciones†† de los cursos ya impartidas; el archivo INFORME_NOTAS contiene las notas obtenidas por los estudiantes en los diversos grupos de los cursos ya impartidos, y el archivo REQUISITO contiene los requisitos previos para cada curso.

Para *definir* esta base de datos, debemos especificar la estructura de los registros de cada archivo indicando los diferentes tipos de **elementos de información** que se almacenarán en cada registro. En la figura 1.2, cada registro ESTUDIANTE incluye datos que representan el Nombre, el NúmEstudiante, el Grado (1, 2, ...) y la Carrera (MATE, ciencias de la computación o CICO, ...) de cada estudiante; cada registro CURSO contiene datos que representan el NombreCurso, el NúmCurso, las HorasCrédito y el Departamento que ofrece el curso, y así sucesivamente. También hay que especificar un **tipo de datos** para cada elemento de información de un registro. Por ejemplo, podemos especificar que Nombre de ESTUDIANTE es una cadena de caracteres alfabéticos, NúmEstudiante de ESTUDIANTE es un entero, y Nota de INFORME_NOTAS es un solo carácter del conjunto {A, B, C, D, F, I}. También podemos representar un elemento de información con un sistema de codificación. Por ejemplo, en la figura 1.2 representamos el Grado de un ESTUDIANTE como un número entre 1 y 4 para un estudiante de licenciatura, o 5 si se trata de uno de posgrado.

Para *construir* la base de datos UNIVERSIDAD, almacenamos datos que representan a cada estudiante, curso, sección, informe de notas de calificación y requisito previo como un registro en el archivo apropiado. Cabe señalar que los registros en los diversos archivos pueden estar relacionados entre sí. Por ejemplo, el registro de "Suárez" en el archivo ESTUDIANTE se relaciona con dos registros del archivo INFORME_NOTAS que especifican las notas de Suárez en dos secciones. De manera similar, cada registro del archivo REQUISITO relaciona dos registros de cursos: uno representa el curso y el otro el requisito previo. En su mayoría, las bases de datos de tamaño mediano y grande contarán con muchos tipos de registros con muchos vínculos entre ellos.

La *manipulación* de la base de datos consiste en las consultas y la actualización. Ejemplos de consultas son "obtener la boleta [una lista de todos los cursos y las notas] de Suárez", "preparar una lista con los nombres de los estudiantes que tomaron la sección del curso de Bases de datos impartida en el otoño de 1992 y sus notas en esa sección", y "¿qué requisitos previos tiene el curso de Bases de datos?". Ejemplos de actualizaciones son "cambiar el grado de Suárez a segundo", "crear una nueva sección del curso de Bases de datos para este semestre", e "introducir una nota de A para Suárez en la sección de Bases de datos del semestre anterior". Estas consultas y actualizaciones informales se deben especificar con precisión en el lenguaje del sistema de base de datos antes de que sean procesadas.

1.3 Características del enfoque de bases de datos

Hay varias características que distinguen el enfoque de bases de datos del enfoque tradicional de programación con archivos. En el **procesamiento de archivos** tradicional, cada usuario define e implementa los archivos requeridos para una aplicación específica. Por ejemplo un usuario, la oficina de informes de notas, podría mantener un archivo de estudiantes y sus respectivas notas, y se escribirían programas para imprimir la boleta de notas de un estudiante y para introducir las nuevas notas en el archivo. Un segundo usuario, la oficina de contabilidad, podría llevar el control de las colegiaturas de los estudiantes y sus pagos. Aunque ambos usuarios están interesados en datos relativos a los estudiantes, cada uno mantiene archivos separados —y programas para manipular dichos archivos— porque requieren datos que no pueden obtener de los archivos del otro. Esta redundancia al definir y almacenar los datos implica espacio de almacenamiento desperdiciado y esfuerzos redundantes para mantener actualizados los datos comunes.

†Definiremos más formalmente las nociones de archivo y registro en el capítulo 4.

††En este ejemplo, y en todos los que aparezcan a lo largo del libro, una "sección de un curso" debe considerarse un grupo atendido por un profesor específico en una fecha determinada. (*N. del R.T.*)

ESTUDIANTE	Nombre	NúmEstudiante	Grado	Carrera
	Suárez	17	1	CICO
	Borja	8	2	CICO

CURSO	NombreCurso	NúmCurso	HorasCrédito	Departamento
	Introd. a la Computación	CICO1310	4	CICO
	Estructura de datos	CICO3320	4	CICO
	Matemáticas discretas	MATE2410	3	MATE
	Base de datos	CICO3380	3	CICO

SECCIÓN	IdentSección	NúmCurso	Semestre	Año	Profesor
	85	MATE2410	Otoño	91	López
	92	CICO1310	Otoño	91	Arreola
	102	CICO3320	Primavera	92	Luján
	112	MATE2410	Otoño	92	Chávez
	119	CICO1310	Otoño	92	Arreola
	135	CICO3380	Otoño	92	Sotelo

INFORME_NOTAS	NúmEstudiante	IdentSección	Notas
	17	112	A
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

REQUISITO	NúmCurso	NúmRequisito
	CICO3380	CICO3320
	CICO3380	MATE2410
	CICO3320	CICO1310

Figura 1.2 Ejemplo de base de datos.

En el enfoque de bases de datos se mantiene un único almacén de datos que se define una sola vez y al cual tienen acceso muchos usuarios. Las principales características del enfoque de bases de datos, en comparación con el de procesamiento de archivos, son las siguientes.

1.3.1 *Naturaleza autodescriptiva de los sistemas de base de datos*

Una característica fundamental del enfoque de bases de datos es que el sistema no sólo contiene la base de datos misma, sino también una definición o descripción completa de la base de datos. Esta definición se almacena en el **catálogo** del sistema, que contiene información como la estructura de cada archivo, el tipo y formato de almacenamiento de cada elemento de información y diversas restricciones que se aplican a los datos. A la información almacenada en el catálogo se le denomina **metadatos**, y éstos describen la estructura de la base de datos primaria (Fig. 1.1).

El catálogo es utilizado por el software del SGBD y, ocasionalmente, por los usuarios de la base de datos que necesitan información sobre la estructura de esta última. El software del SGBD no está escrito para una aplicación de base de datos específica, así que tiene que consultar el catálogo para conocer la estructura de los archivos de una base de datos en particular, como el tipo y el formato de los datos a los que tendrá acceso. El software del SGBD debe trabajar sin menoscabo de su capacidad con *cualquier cantidad de aplicaciones de base de datos* —por ejemplo, bases de datos de una universidad, de algún banco o de una compañía— siempre que la definición de la base de datos esté almacenada en el catálogo.

En el procesamiento de archivos tradicional, la definición de los datos suele ser parte de los programas de aplicación mismos. Por tanto, dichos programas sólo pueden trabajar con *una base de datos específica*, cuya estructura se declara en los programas de aplicación. Por ejemplo, un programa en PASCAL puede incluir declaraciones de estructuras de registros; un programa en C++ puede contener declaraciones "struct" o "class", y un programa en COBOL tiene declaraciones en la división de datos (Data Division) para definir sus archivos. Mientras que el software para el procesamiento de archivos sólo puede tener acceso a bases de datos específicas, el software del SGBD puede acceder a diversas bases de datos al extraer del catálogo las definiciones correspondientes y darles un uso.

En nuestro ejemplo de la figura 1.2, el SGBD almacena en el catálogo las definiciones de todos los archivos mostrados. Siempre que se recibe una solicitud para tener acceso, por ejemplo, al Nombre de un registro ESTUDIANTE, el software del SGBD consultará el catálogo para determinar la estructura del archivo ESTUDIANTE y la posición y el tamaño del elemento Nombre dentro de un registro de ESTUDIANTE. En cambio, en una aplicación de procesamiento de archivos representativa, la estructura del archivo y en algunos casos la ubicación exacta de Nombre dentro de un registro de ESTUDIANTE ya están codificadas en todos los programas que tienen acceso a este elemento de información.

1.3.2 *Separación entre los programas y los datos, y abstracción de los datos*

En el procesamiento de archivos tradicional, la estructura de los archivos de datos viene integrada en los programas de acceso, así que cualquier modificación de la estructura de un archivo puede requerir la *modificación de todos los programas* que tienen acceso a dicho archivo. En cambio, los programas de acceso del SGBD se escriben de modo que sean independientes de cualesquiera archivos específicos. La estructura de los archivos de datos

se almacena en el catálogo del SGBD aparte de los programas de acceso. Llamamos a esta propiedad **independencia con respecto a los programas y datos**. Por ejemplo, se puede escribir un programa de modo que sólo pueda tener acceso a registros de ESTUDIANTE de 42 caracteres de longitud (Fig. 1.3). Si queremos añadir otro dato a cada registro de ESTUDIANTE, digamos FechaNacimiento, un programa así no podrá seguir funcionando y habrá que modificarlo. En contraste, en un entorno de SGBD, basta con modificar la descripción de los registros de ESTUDIANTE en el catálogo, y ningún programa se alterará. La próxima vez que un programa del SGBD consulte el catálogo, tendrá acceso a la nueva estructura de los registros de ESTUDIANTE y la utilizará.

Avances recientes en las bases de datos orientadas a objetos (véase el Cap. 22) y en los lenguajes de programación permiten a los usuarios definir operaciones sobre los datos como parte de las definiciones de la base de datos. Una **operación** (también llamada *función*) se especifica en dos partes. La *interfaz* (o *signatura*) de la operación contiene su nombre y los tipos de datos de sus argumentos (o parámetros). La *implementación* (o *método*) de la operación se especifica aparte y se puede modificar sin afectar la interfaz. Los programas de aplicación de los usuarios pueden operar sobre los datos invocando estas operaciones a través de sus nombres y argumentos, sea cual sea la forma en que se hayan implementado. Esto podría llamarse **independencia con respecto a los programas y operaciones**.

La característica que hace posible la independencia con respecto a los programas y datos y la independencia con respecto a los programas y operaciones se denomina **abstracción de los datos**. Un SGBD ofrece a los usuarios una **representación conceptual** de los datos que no incluye muchos de los detalles de cómo se almacenan. En términos informales, un **modelo de datos** es un tipo de abstracción de los datos con que se obtiene esta representación conceptual. En el modelo de datos intervienen conceptos lógicos, como serían los objetos, sus propiedades y sus interrelaciones, que la mayoría de los usuarios pueden entender más fácilmente que los conceptos de almacenamiento en el computador. Por tanto, el modelo de datos *oculta* los detalles del almacenamiento que no interesan a la mayoría de los usuarios de la base de datos.

Por ejemplo, consideremos la figura 1.2. En una aplicación de procesamiento de archivos, éstos pueden definirse en términos de la longitud de sus registros —el número de caracteres (bytes) que tiene cada uno— y cada elemento de información puede especificarse en términos de su byte inicial dentro de un registro y de su longitud en bytes. Así, un registro de ESTUDIANTE se representaría como en la figura 1.3. Sin embargo, al usuario representativo de la base de datos no le interesa dónde está cada elemento dentro de un registro ni qué longitud tiene; más bien, lo que exigirá es que, cuando se haga referencia a Nombre de ESTUDIANTE, se devuelva el valor correcto. En la figura 1.2 se muestra una representación conceptual de los registros de ESTUDIANTE. El SGBD puede ocultar a los usuarios muchos otros detalles de cómo se organiza el almacenamiento de los archivos, como los caminos de acceso especificados para ellos; estudiaremos los detalles de almacenamiento en los capítulos 4 y 5.

Nombre del dato	Posición inicial en el registro	Longitud en caracteres (bytes)
Nombre	1	30
NúmEstudiante	31	4
Grado	35	4
Carrera	39	4

Figura 1.3 Formato de almacenamiento de un registro ESTUDIANTE.

En el enfoque de bases de datos, la estructura y organización detalladas de todos los archivos se guardan en el catálogo. Los usuarios de la base de datos hacen referencia a la representación conceptual de los archivos, y el SGBD extrae del catálogo los detalles de almacenamiento de éstos cuando los necesita. Hay muchos modelos de datos que sirven para ofrecer a los usuarios esta abstracción de los datos. Una parte importante de este libro se dedica a presentar diversos modelos de datos y los conceptos de que se valen para abstraer la representación de los datos.

Dada la tendencia reciente hacia las bases de datos orientadas a objetos, la abstracción se lleva a un nivel más superior para que contemple no sólo la estructura de los datos, sino también las *operaciones* sobre los datos. Estas operaciones constituyen una abstracción de las actividades del minimundo que casi todos los usuarios entienden. Por ejemplo, se puede aplicar una operación CALCULAR_PDC a un objeto estudiante para calcular el promedio de sus notas. Los programas del usuario pueden invocar tales operaciones aunque él no conozca los detalles de su implementación interna. En este sentido, el usuario hace una abstracción de la actividad del minimundo en forma de una **operación abstracta**.

1.3.3 *Manejo de múltiples vistas de los datos*

Una base de datos suele tener muchos usuarios, cada uno de los cuales puede requerir una perspectiva o **vista** diferente de la mencionada base de datos. Una vista puede ser un subconjunto de la base de datos o contener datos **virtuales** que se deriven de los archivos de la base de datos, pero que no estén almacenados explícitamente. Es posible que algunos usuarios no necesiten saber si los datos a los que hacen referencia están almacenados o son derivados. Un SGBD multiusuario cuyos usuarios tengan diversas aplicaciones debe proporcionar mecanismos para definir muchas vistas. Por ejemplo, puede ser que a un usuario de la base de datos de la figura 1.2 sólo le interesen las boletas de notas de los estudiantes; la vista para este usuario se muestra en la figura 1.4(a). Un segundo usuario, al que sólo le interesa comprobar que los estudiantes hayan tomado los requisitos previos de todos los cursos en los que se inscriben, podría requerir la vista que se muestra en la figura 1.4(b).

1.3.4 *Compartimiento de datos y procesamiento de transacciones multiusuario*

Todo SGBD multiusuario, como su nombre lo indica, debe permitir a varios usuarios tener acceso simultáneo a la base de datos. Esto es indispensable para que los datos de múltiples aplicaciones se integren y mantengan en una sola base de datos. El SGBD debe incluir software de **control de concurrencia** para asegurar que cuando varios usuarios intenten actualizar los mismos datos lo hagan de manera controlada para que el resultado de las actualizaciones sea correcto. Un ejemplo sería el caso de varios encargados de reservaciones que trataran de asignar un asiento en un vuelo comercial; el SGBD debe garantizar que sólo un empleado tenga acceso a un asiento determinado en un momento dado para asignarlo a un pasajero. En general, se dice que éstas son aplicaciones de **procesamiento de transacciones**, y una función fundamental del software del SGBD multiusuario es asegurar que las transacciones concurrentes se realicen de manera correcta sin interferencias.

Las características anteriores son de la mayor importancia cuando se distinga un SGBD del software tradicional de procesamiento de archivos. En la sección 1.6 veremos otras funciones que caracterizan a los SGBD, pero antes clasificaremos los diferentes tipos de personas que trabajan en un entorno de bases de datos.

1.4 Actores en el escenario

En una base de datos personal pequeña, como la lista de direcciones mencionada en la sección 1.1, lo normal es que una sola persona la defina, construya y manipule. En cambio, muchas personas participan en el diseño, uso y mantenimiento de una base de datos grande con algunos cientos de usuarios. En esta sección identificaremos a las personas cuyo trabajo requiere el empleo cotidiano de una base de datos grande; las llamaremos "actores en el escenario". En la sección 1.5 consideraremos a las personas que podrían caer en la categoría de "trabajadores tras bambalinas": quienes laboran para mantener el entorno del sistema de base de datos, pero que no tienen un claro interés en la base de datos en sí misma.

1.4.1 *Administradores de bases de datos*

En cualquier organización en la que muchas personas utilicen los mismos recursos se requiere un administrador en jefe que supervise y controle dichos recursos. En un entorno de bases de datos, el recurso primario es la propia base de datos, y el secundario es el SGBD y el software con él relacionado. La administración de estos recursos es responsabilidad del **administrador de bases de datos** (DBA: *database administrator*). El DBA se encarga de autorizar el acceso a la base de datos, de coordinar y vigilar su empleo, y de adquirir los recursos necesarios de software y hardware. El DBA es la persona responsable cuando surgen problemas como violaciones a la seguridad o una respuesta lenta del sistema. En las organizaciones grandes, el DBA cuenta con la ayuda de un personal para poder desempeñar estas funciones.

1.4.2 *Diseñadores de bases de datos*

Los **diseñadores de bases de datos** se encargan de identificar los datos que se almacenarán en la base de datos y de elegir las estructuras apropiadas para representar y almacenar dichos datos. Por lo general, estas tareas se realizan antes de que de hecho se implemente la base de datos. Los diseñadores tienen la responsabilidad de comunicarse con todos los futuros

(a)

BOLETA	NombreEstudiante	BoletaEstudiante				
		NúmCurso	Notas	Semestre	Año	IdentSección
	Suárez	CICO1310	C	Otoño	92	119
		MATE2410	B	Otoño	92	112
	Borja	MATE2410	A	Otoño	91	85
		CICO1310	A	Otoño	91	92
		CICO3320	B	Primavera	92	102
		CICO3380	A	Otoño	92	135

(b)

REQUISITOS	NombreCurso	NúmCurso	Requisitos
	Base de datos	3380	CICO32320
			MATE2410
	Estructura de datos	3320	CICO1310

Figura 1.4 Dos vistas (datos derivados) de la base de datos de la figura 1.2. (a) La vista de boleta de estudiante. (b) La vista de requisitos previos para los cursos.

usuarios de la base de datos, a fin de comprender sus necesidades, y de presentar un diseño que satisfaga esos requerimientos. En muchos casos los diseñadores forman parte del personal del DBA y tal vez asuman otras responsabilidades una vez terminado el diseño de la base de datos. Casi siempre, los diseñadores interactúan con cada uno de los grupos de usuarios potenciales y desarrollan una **vista** de la base de datos que satisfaga los requerimientos de datos y de procesamiento para ese grupo. Después, se analizan las vistas y se integran con las de otros grupos de usuarios. El diseño final debe ser capaz de satisfacer las necesidades de todos estos grupos.

1.4.3 *Usuarios finales*

Son las personas que necesitan tener acceso a la base de datos para consultarla, actualizarla y generar informes; la base de datos existe primordialmente para que ellos la usen. Hay varias categorías de usuarios finales:

- Los **usuarios finales esporádicos** tienen acceso de vez en cuando a la base de datos, pero es posible que requieran información diferente en cada ocasión. Utilizan un lenguaje de consulta de base de datos avanzado para especificar sus solicitudes, y suelen ser gerentes de nivel medio o alto u otras personas que examinan de modo superficial y ocasional la base de datos.
- Los **usuarios finales simples** o **paramétricos** constituyen una porción apreciable de la totalidad de los usuarios finales. La función principal de su trabajo gira en torno a consultas y actualizaciones constantes de la base de datos, utilizando tipos estándar de estas operaciones —llamadas **transacciones programadas**— que se han programado y probado con mucho cuidado. Todos estamos acostumbrados a tratar con varios tipos de estos usuarios. Los cajeros bancarios revisan saldos y asientan retiros y depósitos. Los encargados de reservaciones de líneas aéreas, hoteles y compañías de alquiler de automóviles revisan las disponibilidades para una solicitud presentada y hacen reservaciones. Los empleados de estaciones receptoras de paquetería introducen las identificaciones de los paquetes por medio de códigos de barras e información descriptiva a través de botones, a fin de actualizar una base de datos centralizada de los paquetes recibidos y los que están en tránsito.
- Entre los **usuarios finales avanzados** se cuentan ingenieros, científicos, analistas de negocios y otros, quienes conocen a cabalidad los recursos del SGBD para satisfacer sus complejos requerimientos.
- Los **usuarios autónomos** emplean bases de datos personalizadas gracias a los paquetes de programas comerciales que cuentan con interfaces de fácil uso, basadas en menús o en gráficos. Un ejemplo es el usuario de un paquete fiscal que almacena diversos datos financieros personales para fines fiscales.

Casi todos los SGBD cuentan con múltiples recursos para tener acceso a una base de datos. Los usuarios finales simples no necesitan aprender mucho sobre los recursos que proporciona el SGBD; sólo tienen que comprender los tipos normales de transacciones diseñadas e implantadas para que ellos las usen. Los usuarios esporádicos aprenden a emplear sólo unos cuantos recursos que quizá utilizarán varias veces. Los usuarios avanzados intentan conocer

la mayor parte de los recursos del SGBD para satisfacer sus complejos requerimientos. Los usuarios autónomos por lo regular adquieren gran habilidad en el uso de un paquete de software específico.

1.4.4 *Analistas de sistemas y programadores de aplicaciones*

Los **analistas de sistemas** determinan los requerimientos de los usuarios finales, sobre todo los de los simples o paramétricos, y desarrollan especificaciones para transacciones programadas que satisfagan dichos requerimientos. Los **programadores de aplicaciones** implementan esas especificaciones en forma de programas, y luego prueban, depuran, documentan y mantienen estas transacciones programadas. Para realizar dichas tareas, estos analistas y programadores deben conocer a la perfección toda la gama de capacidades del SGBD.

1.5 Trabajadores tras bambalinas

Además de los diseñadores, usuarios y administradores de bases de datos, hay otras personas que tienen que ver con el diseño, creación y operación del *software y entorno del sistema* del SGBD. Por lo regular a éstos no les interesa la base de datos misma. Los llamamos trabajadores tras bambalinas y entran en las siguientes categorías.

1.5.1 *Diseñadores e implementadores del SGBD*

Éstos se encargan de diseñar e implementar los módulos e interfaces del SGBD en forma de paquetes de software. Un SGBD es un sistema complejo de software que consta de diversos componentes o **módulos**, como los módulos para implementar el catálogo, los lenguajes de consulta, los procesadores de interfaz, el acceso a los datos y la seguridad. El SGBD debe poder comunicarse con otros programas del sistema, como el sistema operativo y los compiladores de diversos lenguajes de programación.

1.5.2 *Creadores de herramientas*

Las **herramientas** son paquetes de software que facilitan el diseño y el empleo de los sistemas de base de datos, y que ayudan a elevar el rendimiento. Estos paquetes son opcionales y a menudo se adquieren por separado. Incluyen paquetes para diseñar bases de datos, vigilar el rendimiento, proporcionar interfaces de lenguaje natural o de gráficos, elaborar prototipos, realizar simulaciones y generar datos de prueba. Los creadores de herramientas se ocupan de diseñar e implementar estos paquetes. En muchos casos hay proveedores independientes de software, que crean y comercializan estas herramientas.

1.5.3 *Operadores y personal de mantenimiento*

Éstos son los miembros del personal de administración del sistema que tienen a su cargo el funcionamiento y mantenimiento reales del entorno de hardware y software del sistema de base de datos.

Aunque las categorías anteriores de trabajadores tras bambalinas son cruciales para que los usuarios finales puedan servirse del sistema de base de datos, casi nunca utilizan la base de datos para sus propios propósitos.

1.6 Características deseables en un SGBD*

En esta sección analizaremos qué características son deseables en los SGBD y qué capacidades deben ofrecer. El DBA debe aprovechar estas capacidades para lograr diversos objetivos relacionados con el diseño, la administración y el empleo de una base grande de datos multiusuario.

1.6.1 *Control de la redundancia*

En la creación tradicional de programas con procesamiento de archivos, cada grupo de usuarios mantiene sus propios archivos para manejar sus aplicaciones de procesamiento de datos. Por ejemplo, consideremos el ejemplo de la base de datos UNIVERSIDAD de la sección 1.2; ahí, dos grupos de usuarios podrían ser el personal de matriculación a cursos y la oficina de contabilidad. Con el enfoque tradicional, cada grupo mantendría archivos independientes para cada estudiante. La oficina de contabilidad mantendría también datos relacionados con las matriculaciones y la facturación correspondiente, en tanto que la oficina de matriculación controlaría los cursos y las notas de los estudiantes. Una buena parte de los datos se almacenaría dos veces: una vez en los archivos de cada grupo de usuarios. Otros grupos de usuarios podrían duplicar parte de esos datos, o todos, en sus propios archivos.

A veces, y no pocas, esta **redundancia** en el almacenamiento de los mismos datos provoca varios problemas. En primer lugar, es necesario realizar una misma actualización lógica —como introducir los datos de un nuevo estudiante— varias veces: una vez en cada archivo en el que se registren datos de los estudiantes. Esto implica una *duplicación del trabajo*. En segundo lugar, *se desperdicia espacio de almacenamiento* al guardar los mismos datos en varios lugares, y este problema puede ser grave si las bases de datos son grandes. En tercer lugar es posible que los archivos que representan los mismos datos se tornen *inconsistentes*, quizá porque una actualización se haya aplicado a ciertos archivos pero no a otros. Incluso si la actualización —digamos la adición de un nuevo estudiante— se aplica a todos los archivos apropiados, persiste la posibilidad de que los datos relacionados con el estudiante sean **inconsistentes** porque cada grupo de usuarios aplica las actualizaciones de manera independiente. Por ejemplo, un grupo de usuarios podría introducir como fecha de nacimiento del estudiante el valor erróneo 19-ENE-1974, en tanto que los demás grupos de usuarios introducirían el valor correcto 29-ENE-1974.

Con el enfoque de bases de datos, las vistas de los diferentes grupos de usuarios se integran durante el diseño de la base de datos. Para conservar la consistencia, debe crearse un diseño que almacene cada dato lógico —como el nombre o la fecha de nacimiento de un estudiante— en *un solo lugar* de la base de datos. Ello evita la inconsistencia y ahorra espacio de almacenamiento. En algunos casos puede convenir la **redundancia controlada**. Por ejemplo, podríamos almacenar de manera redundante NombreEstudiante y NúmCurso en un archivo INFORME_NOTAS (Fig. 1.5(a)) porque, siempre que recuperemos un registro de INFORME_NOTAS, querremos recuperar el nombre del estudiante y el número del curso junto

(a)

INFORME_NOTAS	NúmEstudiante	NombreEstudiante	IdentSección	NúmCurso	Notas
	17	Suárez	112	MATE2410	B
	17	Suárez	119	CICO1310	C
	8	Borja	85	MATE2410	A
	8	Borja	92	CICO1310	A
	8	Borja	102	CICO3320	B
	8	Borja	135	CICO3380	A

(b)

INFORME_NOTAS	NúmEstudiante	NombreEstudiante	IdentSección	NúmCurso	Notas
	17	Suárez	112	MATE2410	B

Figura 1.5 Almacenamiento redundante de datos en archivos. (a) Redundancia controlada: inclusión de NombreEstudiante y NúmCurso en el archivo INFORME_NOTAS. (b) Redundancia no controlada: un registro de INFORME_NOTAS que es inconsistente con los registros de ESTUDIANTE de la figura 1.2 (el Nombre del estudiante número 17 es Suárez, no Borja).

con la nota, el número del estudiante y el identificador de la sección. Si colocamos juntos todos los datos, no tendremos que buscar en varios archivos los datos que deseamos reunir. En tales casos, el SGBD deberá ser capaz de **controlar** esta redundancia para que no haya inconsistencias entre los archivos. Esto puede lograrse verificando automáticamente que los valores de NombreEstudiante y NúmEstudiante de todo registro de INFORME_NOTAS en la figura 1.5(a) coincidan con los valores de Nombre y NúmEstudiante de un registro de ESTUDIANTE (Fig. 1.2). De manera similar, los valores de IdentSección y NúmCurso de INFORME_NOTAS deberán coincidir con los de algún registro de SECCIÓN. Estas revisiones se pueden especificar durante el diseño de la base de datos, y el SGBD las efectuará siempre que se actualice el archivo INFORME_NOTAS. La figura 1.5(b) muestra un registro de INFORME_NOTAS que no es consistente con el archivo ESTUDIANTE de la figura 1.2, y que podría introducirse erróneamente si no se controla la redundancia.

1.6.2 *Restricción de los accesos no autorizados*

Cuando muchos usuarios comparten una misma base de datos, es probable que no todos tengan la autorización para tener acceso a toda la información que contiene. Por ejemplo, es común considerar que los datos financieros son confidenciales y que sólo ciertas personas puedan tener autorización para tener acceso a ellos. Además, es posible que sólo algunos usuarios tengan permiso para recuperar datos, en tanto que a otros se les permita obtenerlos y actualizarlos. Por tanto, también es preciso controlar el tipo de las operaciones de acceso (obtención o actualización). Por lo regular, a los usuarios o grupos de usuarios se les asignan números de cuenta protegidos con contraseñas, mismos que sirven para tener acceso a la base de datos. El SGBD debe contar con un subsistema de **seguridad y autorización** que permita al DBA crear cuentas y especificar restricciones para ellas. El SGBD deberá entonces obligar automáticamente al cumplimiento de dichas restricciones. Cabe señalar que el mismo tipo de controles se puede aplicar al software del SGBD. Por ejemplo, sólo el personal del DBA tendrá autorización para utilizar cierto software **privilegiado**, como el que sirve para crear cuentas nuevas. De manera similar, podemos hacer que los usuarios paramétricos sólo puedan tener acceso a la base de datos a través de las transacciones programadas que expresamente fueron creadas para ellos.

1.6.3 *Almacenamiento persistente de objetos y estructuras de datos de programas*

Una aplicación reciente de las bases de datos consiste en ofrecer *almacenamiento persistente* para objetos y estructuras de datos de programas. Ésta es una de las principales razones de que se hayan creado los SGBD **orientados a objetos**. Es común que los lenguajes de programación cuenten con estructuras de datos complejas, como los tipos de registro en PASCAL o las definiciones de clases en C++. Los valores de las variables de un programa se desechan una vez que éste termina, a menos que el programador explícitamente los almacene en archivos permanentes; para ello, suele requerirse la conversión de esas estructuras complejas a un formato adecuado para su almacenamiento en archivos. Cuando hay que leer otra vez estos datos, el programador debe convertirlos del formato de archivo a la estructura de variables del programa. Los sistemas de base de datos orientados a objetos son compatibles con lenguajes de programación del tipo de C++ y SMALLTALK, y el software del SGBD realiza automáticamente las conversiones necesarias. Así, podemos almacenar permanentemente un objeto complejo de C++ en un SGBD orientado a objetos, como ObjectStore (véase el Cap. 22). Se dice que los objetos de este tipo son **persistentes** porque sobreviven cuando termina la ejecución del programa y después se pueden recuperar directamente mediante otro programa en C++.

El almacenamiento persistente de objetos y estructuras de datos de programas es una función importante para los sistemas de base de datos. Los SGBD tradicionales a menudo adolecían del llamado *problema de incompatibilidad de impedancia* porque las estructuras de datos proporcionadas por el SGBD eran incompatibles con las del lenguaje de programación. Los sistemas de base de datos orientados a objetos suelen ofrecer *compatibilidad* de las estructuras de datos con uno o más lenguajes de programación orientada a objetos.

1.6.4 *Inferencias en la base de datos mediante reglas de deducción*

Otra aplicación reciente de los sistemas de base de datos consiste en ofrecer recursos para definir *reglas de deducción* que permitan *deducir* o *inferir* información nueva a partir de los datos almacenados. A estos sistemas se les conoce como bases de datos **deductivas**. Por ejemplo, puede haber reglas complejas en la aplicación del minimundo para determinar cuándo un estudiante está a prueba. Estas reglas se pueden especificar *de manera declarativa* como reglas de deducción, con cuya aplicación será posible determinar cuáles estudiantes están a prueba. En un SGBD tradicional se tendría que escribir un *programa por procedimientos* explícito para apoyar tales aplicaciones. Pero si cambian las reglas del minimundo, casi siempre es más fácil modificar las reglas de deducción declaradas que volver a codificar los programas por procedimientos.

1.6.5 *Suministro de múltiples interfaces con los usuarios*

En vista de que muchos tipos de usuarios con diversos niveles de conocimientos técnicos utilizan las bases de datos, el SGBD debe ofrecer diferentes interfaces. Entre éstas podemos mencionar los lenguajes de consulta para usuarios esporádicos, las interfaces de lenguaje de programación para programadores de aplicaciones, las formas y códigos de órdenes para los usuarios paramétricos y las interfaces controladas por menús y en lenguaje natural para los usuarios autónomos.

1.6.6 *Representación de vínculos complejos entre los datos*

Una base de datos puede contener numerosos conjuntos de datos que estén relacionados entre sí de muchas maneras. Consideremos el ejemplo de la figura 1.2. El registro de Borja en el archivo ESTUDIANTE se relaciona con cuatro registros del archivo INFORME_NOTAS. De manera similar, cada registro de SECCIÓN se relaciona con un registro de CURSO y también con varios registros de INFORME_NOTAS, uno por cada estudiante que haya concluido esa sección. Es preciso que el SGBD pueda representar diversos vínculos complejos de los datos y también obtener y actualizar con rapidez y eficiencia datos que estén mutuamente relacionados.

1.6.7 *Cumplimiento de las restricciones de integridad*

La mayor parte de las aplicaciones de base de datos tienen ciertas **restricciones de integridad** que deben cumplir los datos. El SGBD debe ofrecer recursos para definir tales restricciones y hacer que se cumplan. La forma más simple de restringir la integridad consiste en especificar un tipo de datos para cada elemento de información. Por ejemplo, en la figura 1.2 podemos especificar que el valor del elemento Grado dentro de cada registro de ESTUDIANTE debe ser un entero entre 1 y 5, y que el valor de Nombre debe ser una cadena de no más de 30 caracteres alfabéticos. Otro tipo de restricción que encontramos a menudo, más complejo, implica especificar que un registro de un archivo debe relacionarse con registros de otros archivos. Por ejemplo, en la figura 1.2 podemos especificar que "todo registro de SECCIÓN debe estar relacionado con un registro de CURSO". Otro tipo de restricción especifica que los valores de los elementos de información sean únicos; por ejemplo, "cada registro de CURSO debe tener un valor único de NúmCurso". Estas restricciones se derivan de la **semántica** (o significado) de los datos y del minimundo que representa. Es responsabilidad de los diseñadores de la base de datos identificar las restricciones de integridad durante el diseño. Algunas restricciones se pueden especificar en el SGBD, el cual hará automáticamente que se cumplan; otras pueden requerir verificación mediante programas de actualización o en el momento en que se introducen los datos.

Es posible introducir erróneamente un dato sin violar las restricciones de integridad. Por ejemplo, si un estudiante obtiene una nota de A pero se introduce C en la base de datos, el SGBD *no podrá* descubrir este error automáticamente, porque C es un valor permitido del tipo de datos de Notas. Esta clase de errores sólo puede descubrirse manualmente (cuando el estudiante reciba su boleta de notas y se queje) y corregirse después actualizando la base de datos. Por otro lado, el SGBD puede rechazar automáticamente una nota de X, porque éste no es un valor permitido para el tipo de datos de Notas.

1.6.8 *Respaldo y recuperación*

Todo SGBD debe contar con recursos para recuperarse de fallos de hardware o de software. Para ello está el subsistema de **respaldo y recuperación** del SGBD. Por ejemplo, si el sistema falla mientras se está ejecutando un complejo programa de actualización, el subsistema de recuperación se encargará de asegurarse de que la base de datos se restaure al estado en el que estaba antes de que comenzara la ejecución del programa. Como alternativa, el subsistema de recuperación puede asegurarse de que el programa reanude su ejecución en el punto en que fue interrumpido, de modo que su efecto completo se registre en la base de datos.

1.7 Implicaciones del enfoque de bases de datos*

Además de los aspectos analizados en la sección anterior, hay otras implicaciones del empleo del enfoque de bases de datos que pueden resultar provechosas para casi todas las organizaciones.

1.7.1 *Potencial para imponer normas*

Con el enfoque de las bases de datos el DBA puede definir e imponer normas a los usuarios de la base de datos en una organización grande. Esto facilita la comunicación y cooperación entre diversos departamentos, proyectos y usuarios de esa organización. Es posible definir normas para los nombres y formatos de los elementos de información, para los formatos de presentación, para las estructuras de los informes, para la terminología, etc. Es más fácil que el DBA imponga normas en un entorno centralizado de base de datos que en un entorno en el que cada grupo de usuarios tenga el control de sus propios archivos y programas.

1.7.2 *Menor tiempo de creación de aplicaciones*

Una de las características más convincentes a favor del enfoque de bases de datos es que la creación de una aplicación nueva —como la obtención de cierta información de la base de datos para imprimir un nuevo informe— requiere muy poco tiempo. Diseñar e implementar una nueva base de datos desde cero puede tardar más que escribir una sola aplicación de archivos especializada; sin embargo, una vez que la base de datos está construida y en funciones, casi siempre se requerirá mucho menos tiempo para crear nuevas aplicaciones con los recursos del SGBD. Se estima que el tiempo de creación con un SGBD es de una sexta a una cuarta parte del requerido en un sistema de archivos tradicional.

1.7.3 *Flexibilidad*

En ocasiones es necesario modificar la estructura de una base de datos cuando cambian los requerimientos. Por ejemplo, podría surgir un nuevo grupo de usuarios que necesite información adicional que no se encuentra actualmente en la base de datos. Para atenderlos, tal vez sea necesario añadir un nuevo archivo a la base de datos o extender los elementos de un archivo ya existente. Algunos SGBD permiten efectuar estas modificaciones en la estructura de la base de datos sin afectar los datos almacenados y los programas de aplicación ya existentes.

1.7.4 *Disponibilidad de información actualizada*

Los SGBD ponen la base de datos a disposición de todos los usuarios. En el momento en que un usuario actualiza la base de datos, todos los demás usuarios pueden ver de inmediato esta actualización. Esta disponibilidad de información actualizada es indispensable en muchas aplicaciones de procesamiento de transacciones, como los sistemas de reservaciones o las bases de datos bancarias, y se hace posible gracias a los subsistemas de control de concurrencia y de recuperación del SGBD.

1.7.5 *Economías de escala*

El enfoque de SGBD permite consolidar los datos y las aplicaciones, reduciéndose así el desperdicio por traslapo entre las actividades del personal de procesamiento de datos en los diferentes proyectos o departamentos. Esto permite que la organización completa invierta en procesadores más potentes, dispositivos de almacenamiento o equipo de comunicación, en vez de que cada departamento tenga que adquirir por separado su propio equipo (de menor capacidad). Esto reduce los costos totales de operación y control.

1.8 Cuándo no usar un SGBD*

A pesar de todas estas ventajas, hay situaciones en las que el empleo de un SGBD puede generar costos adicionales innecesarios que se evitarían con el procesamiento de archivos tradicional. Hay varias causas de estos costos adicionales por utilizar un SGBD, entre ellas:

- Una fuerte inversión inicial en equipo, software y capacitación.
- La generalidad que ofrece el SGBD para definir y procesar los datos.
- Los costos que implica ofrecer las funciones de seguridad, control de concurrencia, recuperación e integridad.

Pueden surgir problemas adicionales si los diseñadores de la base de datos y el DBA no producen un diseño adecuado o si la implementación de las aplicaciones del sistema de base de datos no es correcta. En vista de los costos adicionales de emplear un SGBD y de los problemas potenciales de una administración inadecuada, puede ser más conveniente utilizar archivos ordinarios en las siguientes circunstancias:

- La base de datos y las aplicaciones son simples, están bien definidas, y no se espera que cambien.
- Algunos programas tienen requerimientos estrictos de tiempo real que no podrían cumplirse por el costo extra del SGBD.
- No se requiere acceso multiusuario a los datos.

1.9 Resumen

En este capítulo definimos una base de datos como un conjunto de datos relacionados entre sí, donde *datos* significa hechos registrados. Por lo regular, una base de datos representa algún aspecto del mundo real, y sirve para fines específicos de uno o más grupos de usuarios. Un SGBD consiste en un software generalizado para implementar y mantener una base de datos computarizada. La base de datos y el software, en conjunto, constituyen un sistema de base de datos. Identificamos varias características con que se distinguen el enfoque de bases de datos y las aplicaciones tradicionales de procesamiento de archivos:

- Existencia de un catálogo o diccionario de datos.
- Abstracción de los datos.

- Independencia con respecto a programas y datos, y con respecto a programas y operaciones.
- Manejo de múltiples vistas de usuarios.
- Compartimiento de datos entre varias transacciones.

Después examinamos las principales categorías de usuarios que tendrá la base de datos, los "actores en el escenario":

- Diseñadores y administradores de bases de datos.
- Usuarios finales.
- Programadores de aplicaciones y analistas de sistemas.

Señalamos que, además de los usuarios, hay varias categorías de personal de apoyo, o "trabajadores tras bambalinas", en un entorno de bases de datos:

- Diseñadores e implementadores de SGBD.
- Creadores de herramientas.
- Operadores y personal de mantenimiento.

Luego presentamos una lista de los recursos de ayuda que el software del SGBD debe ofrecer al DBA, a los diseñadores de bases de datos y a los usuarios para administrar, diseñar y utilizar una base de datos:

- Control de redundancia.
- Restricción de accesos no autorizados.
- Almacenamiento persistente para estructuras de datos de programas.
- Inferencias que permiten generar las reglas de deducción.
- Múltiples interfaces.
- Representación de vínculos complejos entre los datos.
- Imposición de restricciones de integridad.
- Respaldo y recuperación.

Mencionamos algunas ventajas adicionales que ofrece el enfoque de bases de datos y que no tienen los sistemas tradicionales de procesamiento de archivos:

- Potencial para imponer normas.
- Flexibilidad.
- Más rapidez para crear aplicaciones.
- Disponibilidad de información actualizada para todos los usuarios.
- Economías de escala.

Por último, analizamos los costos adicionales que ocasiona el empleo de un SGBD y mencionamos algunas situaciones en las que podría ser más conveniente no utilizarlo. En la cronología que aparece al final del capítulo se ofrece un panorama general de los principales avances en el campo de las bases de datos.

Preguntas de repaso

1.1. Defina los siguientes términos: *datos*, *base de datos*, SGBD, *sistema de base de datos*, *catálogo de base de datos*, *independencia con respecto a programas y datos*, *vista de usuario*, DBA, *usuario final*, *transacción programada*, *sistema de base de datos deductiva*, *objeto persistente*, *metadatos*, *aplicación de procesamiento de transacciones*.

1.2. ¿Cuáles son los tres tipos principales de acciones en las que intervienen las bases de datos? Analice brevemente cada uno de ellos.

1.3. Analice las características principales del enfoque de bases de datos y sus diferencias con respecto a los sistemas tradicionales de archivos.

1.4. ¿Cuáles son las responsabilidades del DBA y de los diseñadores de bases de datos?

1.5. Cite los diferentes tipos de usuarios finales de las bases de datos y analice las actividades principales de cada uno de ellos.

1.6. Analice los recursos con que debe contar todo SGBD.

Ejercicios

1.7. Mencione algunas consultas informales y operaciones de actualización que sería lógico aplicar a la base de datos de la figura 1.2.

1.8. ¿Qué diferencia hay entre la redundancia controlada y la no controlada? Ilústrela con ejemplos.

1.9. Identifique todos los vínculos entre los registros de la base de datos que se muestra en la figura 1.2.

1.10. Mencione algunas vistas adicionales que podrían requerir otros grupos de usuarios de la base de datos de la figura 1.2.

1.11. Cite ejemplos de restricciones de integridad que deban cumplirse, en su opinión, en la base de datos de la figura 1.2.

Bibliografía selecta

El número de marzo de 1976 de ACM *Computing Surveys* contiene una introducción básica a los modelos de datos tradicionales. En dicho número, Sibley (1976) y Fry y Sibley (1976) explican en perspectiva cómo evolucionaron las bases de datos a partir del procesamiento de archivos tradicional. En el número de octubre de 1991 de *Communications of the* ACM aparecen varios artículos que describen los SGBD de la "siguiente generación". Muchos textos sobre bases de datos tratan el material presentado aquí y en el capítulo 2, entre ellos Wiederhold (1986), Ullman (1988), Date (1990) y Korth y Silberschatz (1991).

SISTEMAS DE BASES DE DATOS: UNA BREVE CRONOLOGÍA

Acontecimiento	Consecuencia
Antes de 1960	
1945 Invención de las cintas magnéticas (primer medio que permite búsquedas)	Sustituyeron a las tarjetas perforadas y las cintas de papel.
1957 Instalación del primer computador comercial.	
1959 McGee propone el concepto de acceso generalizado a datos almacenados electrónicamente.	
1959 IBM presenta el sistema Ramac.	Leía datos en forma no secuencial, haciendo factible el acceso a los archivos.
Los años sesenta	
1961 Bachman diseña el primer SGBD generalizado, el almacén de datos integrados (Integrated Data Store, IDS) de GE; amplia distribución hacia 1964. Bachman popularizó los diagramas de estructuras de datos.	Constituyó el fundamento para el modelo de datos de red desarrollado por el Conference on Data Systems Languages Database Task Group (CODASYL DBTG, grupo de trabajo sobre bases de datos de la conferencia sobre lenguajes de sistemas de datos).
1965-1970 • Muchos proveedores crean sistemas generalizados de manejo de archivos.	Ofrecían una vista en dos niveles, conceptual y del usuario, de la organización de los datos. Constituyó el fundamento para el modelo de datos jerárquico.
• IBM desarrolla su Sistema de gestión de información (Information Management System, IMS).	Manejaba vistas de red superpuestas a las jerárquicas.
• El sistema IMS DB/DC (base de datos/ comunicación de datos) fue el primer sistema DB/DC a gran escala.	Permitía el acceso de múltiples usuarios a los datos a través de una red de comunicaciones.
• IBM y American Airlines crean SABRE.	
Los años setenta	
La tecnología de bases de datos experimenta un rápido crecimiento.	Los sistemas comerciales siguieron la propuesta CODASYL DBTG, pero ninguno la implementó por completo. Sistema IDMS de B.F. Goodrich, IDS-II de Honeywell, DMS 1100 de UNIVAC, DMS-II de Burroughs, DMS-170 de CDC, PHOLAS de Phillips y DBMS-11 de Digital.
	Varios sistemas integrados DB/DC. TOTAL de Cincom, y también ENVIRON/1. Los SGBD se establecen como disciplina académica y área de investigación.
1970 Ted Codd, investigador asociado de IBM, desarrolla el modelo relacional.	Estableció los fundamentos para la teoría de bases de datos.
1971 Informe del grupo de trabajo sobre bases de datos (DBTG) de CODASYL.	

1975 El Special Interest Group on Management of Data (Grupo de interés especial de la ACM dedicado a gestión de datos) organiza la primera conferencia internacional SIGMOD.	Constituyó un foro para diseminar las investigaciones sobre bases de datos.
1975 La Very Large Data Base Foundation (Fundación para bases de datos muy grandes) organizó la primera conferencia internacional sobre bases de datos muy grandes (VLDB).	Estableció otro foro para la propagación de las investigaciones sobre bases de datos.
1976 Chen introduce el modelo de entidad-vínculo (ER).	
• **Proyectos de investigación en los años setenta:** System R (IBM), INGRES (University of California, Berkeley), System 2000 (University of Texas, Austin), proyecto Socrate (Universidad de Grenoble, Francia), ADABAS (Universidad Técnica de Darmstadt, Alemania Occ.).	
• **Lenguajes de consulta desarrollados en los años setenta:** SQUARE, SEQUEL (SQL), QBE, QUEL.	
Los años ochenta	
Se desarrollan SGBD para computadores personales (DBASE, PARADOX, etc.).	Permitieron a los usuarios de PC definir y manipular datos. Carecían de recursos para multivista/multiacceso y de separación entre programas y datos.
1983 Estudio de ANSI/SPARC revela que se habían implementado más de 100 sistemas relacionales a principios de los años ochenta.	Aparición de SGBD relacionales comerciales (DB2, ORACLE, SYBASE, INFORMIX, etc.).
1985 Se publica la norma preliminar de SQL. Influencia de los "lenguajes de cuarta generación" en el mundo de los negocios. ANSI propone un lenguaje de definición de redes (NDL: Network Definition Language).	Generaron programas de aplicación completos partiendo de una interfaz de lenguaje de alto nivel para no programadores.
• **Tendencias en los años ochenta**: Sistemas expertos de base de datos, SGBD orientados a objetos, arquitectura cliente-servidor para bases de datos distribuidas.	Permitieron nuevas aplicaciones de las bases de datos, trabajo con redes y gestión de datos distribuidos.
Los años noventa	
• Demanda para extender las capacidades de los SGBD para nuevas aplicaciones.	Características de SGBD para datos espaciales, temporales y de multimedia, incorporando capacidades activas y deductivas.
• Aparición de SGBD comerciales orientados a objetos.	
• Demanda de aplicaciones que utilicen datos de diversas fuentes.	Aparición de normas para consulta e intercambio de datos (SQL2, PDES, STEP); extensión de las capacidades de los SGBD a sistemas heterogéneos y multibases de datos.
• Demanda para aprovechar procesadores paralelos masivos (MPP).	Mejoró el rendimiento de los SGBD comerciales.

CAPÍTULO 2

Conceptos y arquitectura de los sistemas de base de datos

En este capítulo estudiaremos con mayor detalle muchos de los conceptos y problemas vistos en el capítulo 1. También presentaremos la terminología que valdrá para todo el libro. Comenzaremos en la sección 2.1 con un análisis de los modelos de datos y con la definición de los conceptos de esquemas y ejemplares, fundamentales para estudiar los sistemas de base de datos. A continuación trataremos la arquitectura de los SGBD y la independencia con respecto a los datos, en la sección 2.2; los diferentes tipos de interfaces y lenguajes que proporcionan los SGBD, en la sección 2.3; el entorno de software de los sistemas de bases de datos, en la sección 2.4, y la clasificación de los SGBD, en la sección 2.5.

El lector puede examinar superficialmente o saltarse el material de las secciones 2.4 y 2.5.

2.1 Modelos de datos, esquemas y ejemplares

Una característica fundamental del enfoque de bases de datos es que proporciona cierto nivel de abstracción de los datos al ocultar detalles de almacenamiento que la mayoría de los usuarios no necesitan conocer. Los modelos de datos son el principal instrumento para ofrecer dicha abstracción. Un **modelo de datos** es un conjunto de conceptos que pueden servir para describir la estructura de una base de datos.† Con el concepto de *estructura de*

†En ocasiones se utiliza la palabra *modelo* para denotar una descripción, o esquema, de una base de datos: por ejemplo, "el modelo de datos de comercialización". No adoptaremos aquí esta interpretación.

una base de datos nos referimos a los tipos de datos, los vínculos y las restricciones que deben cumplirse para esos datos. Por lo regular, los modelos de datos contienen además un conjunto de **operaciones básicas** para especificar lecturas y actualizaciones de la base de datos. Cada vez se difunde más la práctica de incluir en el modelo de datos conceptos para especificar **comportamiento**; esto es, especificar un conjunto de **operaciones definidas por el usuario** que sean válidas para la base de datos, además de las operaciones básicas incluidas en el modelo de datos. Un ejemplo de operación definida por el usuario es CALCULAR_PDC, que se puede aplicar a un objeto ESTUDIANTE. Por otro lado, casi siempre el modelo de datos básico cuenta con operaciones genéricas para insertar, eliminar, modificar o recuperar un objeto.

2.1.1 *Categorías de los modelos de datos*

Se han propuesto muchos modelos de datos, y podemos clasificarlos dependiendo de los tipos de conceptos que ofrecen para describir la estructura de la base de datos. Los modelos de datos **de alto nivel** o **conceptuales** disponen de conceptos muy cercanos al modo como la generalidad de los usuarios percibe los datos, en tanto que los modelos de datos **de bajo nivel** o **físicos** proporcionan conceptos que describen los detalles de cómo se almacenan los datos en el computador. Los conceptos de los modelos de datos de bajo nivel casi siempre están dirigidos a los especialistas en computación, no a los usuarios finales corrientes. Entre estos dos extremos hay una clase de modelos de datos **de representación** (o **de implementación**), cuyos conceptos pueden ser entendidos por los usuarios finales aunque no están demasiado alejados de la forma en que los datos se organizan dentro del computador. Los modelos de datos de representación ocultan algunos detalles de cómo se almacenan los datos, pero pueden implementarse de manera directa en un sistema de computador.

Los modelos de datos de alto nivel utilizan conceptos como entidades, atributos y vínculos.† Una **entidad** representa un objeto o concepto del mundo real, como un empleado o un proyecto, almacenado en la base de datos. Un **atributo** representa alguna propiedad de interés que da una descripción más amplia de una entidad, como el nombre o el salario del empleado. Un **vínculo** describe una interacción entre dos o más entidades; por ejemplo, el vínculo de trabajo ("trabaja en") entre un empleado y un proyecto. En el capítulo 3 presentaremos el modelo de entidad-vínculo (ER), que es un modelo de datos de alto nivel muy popular.

Los modelos de datos de representación o de implementación son los más utilizados en los SGBD comerciales actuales, y entre ellos se cuentan los tres modelos más comunes: el relacional, el de red y el jerárquico. Las partes II y III del libro describen estos modelos, sus operaciones y sus lenguajes. Representan los datos valiéndose de estructuras de registros, por lo que a veces se les denomina modelos de datos **basados en registros**. Podemos concebir a los modelos de datos **orientados a objetos** como una nueva familia de modelos de implementación de alto nivel más próxima a los modelos conceptuales. En el capítulo 22 describiremos las características generales de estos modelos, así como dos SGBD orientados a objetos.

†En todo este libro aparecen dos términos fundamentales: *relation* y *relationship*. Sin embargo, la traducción de ambos al español es la palabra "relación", y es así como aparece en muchos textos, tanto al hablar del "modelo de entidad-relación" (*relationship*) como del "modelo relacional" (*relation*). No obstante, como en muchas partes del libro se tratan ambos modelos simultáneamente, hemos considerado indispensable distinguir entre los dos términos, adoptando la palabra "relación" para *relation* y la palabra "vínculo" para *relationship*. (*N. del T.*)

También es frecuente utilizar los modelos orientados a objetos como modelos conceptuales de alto nivel, sobre todo en el área de la ingeniería de software. En el capítulo 21 analizaremos otros enfoques del modelado de datos de alto nivel.

Los modelos de datos físicos describen cómo se almacenan los datos en el computador, al representar información como los formatos y ordenamientos de los registros y los caminos de acceso. Un **camino de acceso** es una estructura que hace eficiente la búsqueda de registros específicos de la base de datos. Hablaremos de las técnicas de almacenamiento físico y de las estructuras de acceso en los capítulos 4 y 5.

2.1.2 *Esquemas y ejemplares*

En cualquier modelo de datos es importante distinguir entre la *descripción* de la base de datos y la *base de datos misma*. La descripción se conoce como **esquema de la base de datos** (o **metadatos**). Este esquema se especifica durante el diseño y no es de esperar que se modifique muy a menudo. En la mayoría de los modelos de datos se utilizan ciertas convenciones para representar los esquemas en forma de diagramas, así que la representación de un esquema se denomina **diagrama del esquema**. En la figura 2.1 se muestra un diagrama esquemático de la base de datos de la figura 1.2; el diagrama presenta la estructura de todos los tipos de registros pero no los ejemplares reales de los registros. A cada uno de los objetos del esquema —como ESTUDIANTE o CURSO— se le llama **elemento del esquema**.

Los diagramas de esquema sólo ilustran *algunos aspectos* del esquema, como los nombres de los tipos de registros y de los elementos de información, y algunas clases de restricciones. No se especifican otros aspectos en los diagramas del esquema; por ejemplo, la figura 2.1 no muestra el tipo de datos de cada elemento de información ni los vínculos entre los diferentes archivos. Muchos tipos de restricciones no se representan en los diagramas de esquema; por ejemplo, es bastante difícil representar una restricción como "los estudiantes de la carrera de ciencias de la computación deben cursar CICO1310 antes de terminar el segundo año".

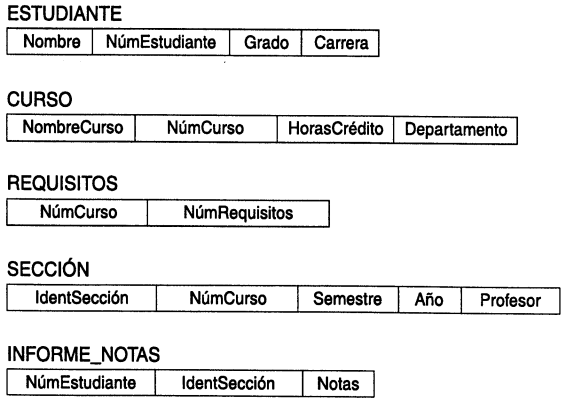


Figura 2.1 Diagrama de esquema para la base de datos de la figura 1.2.

Los datos reales de la base de datos pueden cambiar con mucha frecuencia; por ejemplo, la base de datos de la figura 1.2 cambia cada vez que agregamos un nuevo estudiante o introducimos una nueva nota de algún estudiante. Los datos que la base de datos contiene en un determinado momento se denominan **estado de la base de datos** (o conjunto de **ocurrencias** o **ejemplares**). En un estado dado de la base de datos, cada elemento del esquema tiene su propio *conjunto actual* de ejemplares; por ejemplo, el elemento ESTUDIANTE contendrá como ejemplares el conjunto de entidades estudiante individuales (registros). Es posible construir muchos estados de la base de datos que correspondan a un esquema en particular. Cada vez que insertamos o eliminamos un registro, o que modificamos el valor de un elemento de información, transformamos un estado de la base de datos en otro.

La distinción entre el esquema y el estado de la base de datos es muy importante. Cuando **definimos** una nueva base de datos, sólo especificamos su esquema al SGBD. En ese momento, el estado correspondiente de la base de datos es el "estado vacío", sin datos. Cuando **cargamos** éstos por primera vez, la base de datos pasa al "estado inicial". De ahí en adelante, siempre que se aplique una operación de actualización a la base de datos, tendremos otro estado. El SGBD se encarga en parte de asegurar que *todos* los estados de la base de datos sean **estados válidos**; esto es, que satisfagan la estructura y las restricciones especificadas en el esquema. Por tanto, es en extremo importante especificar un esquema correcto para el SGBD, y debemos tener muchísimo cuidado al diseñarlo. El SGBD almacena el esquema en su catálogo, de modo que el software del SGBD pueda consultarlo siempre que necesite hacerlo. En ocasiones, al esquema se le llama **intensión**, y a un estado de la base de datos, **extensión** del esquema.

2.2 Arquitectura de un SGBD e independencia con respecto a los datos

Hay tres características importantes inherentes al enfoque de las bases de datos, mencionadas en la sección 1.3, que son (a) la separación entre los programas y los datos (independencia con respecto a los programas y datos y con respecto a los programas y operaciones); (b) el manejo de múltiples vistas de usuario, y (c) el empleo de un catálogo para almacenar la descripción (esquema) de la base de datos. En esta sección especificaremos una arquitectura para los sistemas de bases de datos, denominada **arquitectura de tres esquemas**,[†] propuesta como ayuda para contar con estas características. Después, analizaremos el concepto de independencia con respecto a los datos.

2.2.1 *La arquitectura de tres esquemas*

El objetivo de la arquitectura de tres esquemas, que ilustramos en la figura 2.2, consiste en formar una separación entre las aplicaciones del usuario y la base de datos física. En esta arquitectura, los esquemas se pueden definir en los tres niveles siguientes:

[†]También se le conoce como arquitectura ANSI/SPARC, por el comité que la propuso (Tsichritzis y Klug 1978).

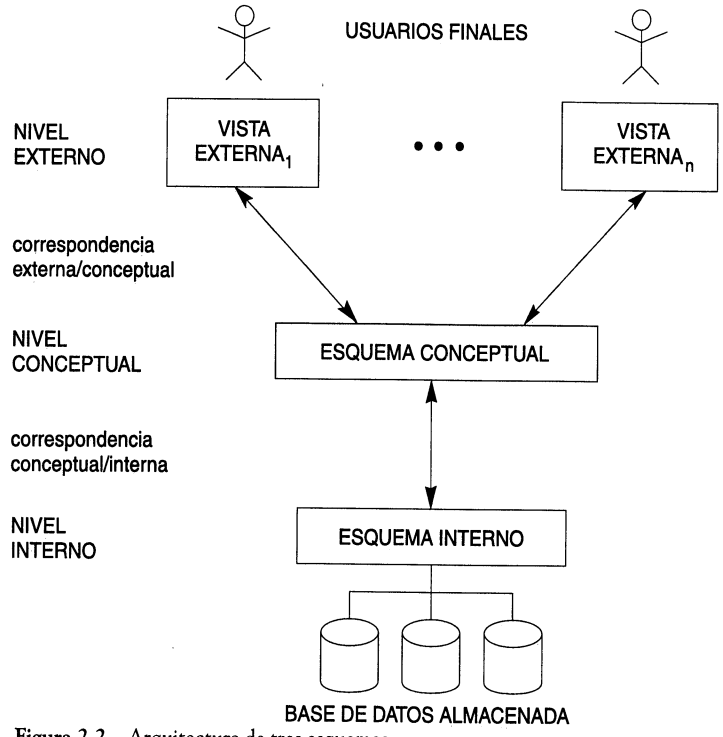


Figura 2.2 Arquitectura de tres esquemas.

1. El **nivel interno** tiene un **esquema interno**, que describe la estructura física de almacenamiento de la base de datos. El esquema interno emplea un modelo físico de los datos y describe todos los detalles para su almacenamiento, así como los caminos de acceso para la base de datos.
2. El **nivel conceptual** tiene un **esquema conceptual**, que describe la estructura de toda la base de datos para una comunidad de usuarios. El esquema conceptual oculta los detalles de las estructuras físicas de almacenamiento y se concentra en describir entidades, tipos de datos, vínculos, operaciones de los usuarios y restricciones. En este nivel podemos usar un modelo de datos de alto nivel o uno de implementación.
3. El **nivel externo** o **de vistas** incluye varios **esquemas externos** o **vistas de usuario**. Cada esquema externo describe la parte de la base de datos que interesa a un grupo de usuarios determinado, y oculta a ese grupo el resto de la base de datos. En este nivel podemos usar un modelo de datos de alto nivel o uno de implementación.

En su mayoría, en los SGBD no se distinguen del todo los tres niveles, pero en algunos de ellos se cuenta, en cierta medida, con la arquitectura de tres esquemas. Algunos SGBD incluyen detalles del nivel físico en el esquema conceptual. En casi todos los SGBD que manejan vistas de usuario, los esquemas externos se especifican en el mismo modelo de datos que describe la información a nivel conceptual. Con algunos SGBD es posible utilizar diferentes modelos de datos en los niveles conceptual y externo.

Cabe señalar que los tres esquemas no son más que descripciones de los datos; los únicos datos que existen *realmente* están en el nivel físico. En un SGBD basado en la arquitectura de tres esquemas, cada grupo de usuarios hace referencia exclusivamente a su propio esquema externo; por tanto, el SGBD debe transformar una solicitud expresada en términos de un esquema externo a una solicitud expresada en términos del esquema conceptual, y luego a una solicitud en el esquema interno que se procesará sobre la base de datos almacenada. Si la solicitud es una obtención de datos, será preciso modificar el formato de la información extraída de la base de datos almacenada para que coincida con la vista externa del usuario. El proceso de transformar solicitudes y resultados de un nivel a otro se denomina **correspondencia** o **transformación** (*mapping*). Estas correspondencias pueden requerir bastante tiempo, por lo que algunos SGBD —sobre todo los que deben manejar bases de datos pequeñas— no cuentan con vistas externas. Sin embargo, incluso en tales sistemas, es preciso realizar algunas correspondencias para transformar solicitudes entre los niveles conceptual e interno.

2.2.2 *Independencia con respecto a los datos*

La arquitectura de tres esquemas puede servir para explicar el concepto de **independencia con respecto a los datos**, que podemos definir como la capacidad para modificar el esquema en un nivel del sistema de base de datos sin tener que modificar el esquema del nivel inmediato superior. Podemos definir dos tipos de independencia con respecto a los datos:

1. La **independencia lógica con respecto a los datos** es la capacidad de modificar el esquema conceptual sin tener que alterar los esquemas externos ni los programas de aplicación. Podemos modificar el esquema conceptual para ampliar la base de datos (añadiendo un nuevo tipo de registro o un elemento de información), o para reducir la base de datos (eliminando un tipo de registro o un elemento de información). En el segundo caso, la modificación no deberá afectar los esquemas externos que sólo se refieran a los datos restantes. Por ejemplo, el esquema externo de la figura 1.4(a) no deberá alterarse si se modifica el archivo INFORME_NOTAS de la figura 1.2 para convertirlo en el que aparece en la figura 1.5(a). Si en el SGBD se cuenta con independencia lógica con respecto a los datos, sólo será preciso modificar la definición de la vista y las correspondencias. Los programas de aplicación que hagan referencia a los elementos del esquema externo deberán funcionar igual que antes después de una reorganización lógica del esquema conceptual. Además, las restricciones podrán modificarse en el esquema conceptual sin afectar los esquemas externos.
2. La **independencia física con respecto a los datos** es la capacidad de modificar el esquema interno sin tener que alterar el esquema conceptual (o los externos). Tal vez sea preciso modificar el esquema interno por la necesidad de reorganizar ciertos archivos físicos —por ejemplo, al crear estructuras de acceso adicionales— a fin de mejorar el rendimiento de las operaciones de obtención o actualización. Si la base de datos aún contiene los mismos datos, no deberá ser necesario modificar el esquema conceptual. Por ejemplo, si se establece un camino de acceso para agilizar la obtención de los registros de SECCIÓN (Fig. 1.2) por Semestre y Año, no será necesario modificar una consulta como "listar todas las secciones que se ofrecieron en el otoño de 1991", aunque el SGBD podrá ejecutarla con mayor eficacia si aprovecha el nuevo

camino de acceso. Dado que la *independencia física con respecto a los datos* se refiere sólo a la separación entre las aplicaciones y las estructuras físicas de almacenamiento, es más fácil de lograr que la independencia lógica con respecto a los datos.

En todo SGBD de múltiples niveles es preciso ampliar el catálogo de modo que incluya información sobre cómo establecer la correspondencia entre las solicitudes y los datos entre los diversos niveles. El SGBD utiliza software adicional para realizar estas correspondencias haciendo referencia a la información de correspondencia que se encuentra en el catálogo. La independencia con respecto a los datos se logra porque, al modificarse el esquema en algún nivel, el esquema del nivel inmediato superior permanece sin cambios; sólo se modifica la correspondencia entre los dos niveles. No es preciso modificar los programas de aplicación que hacen referencia al esquema del nivel superior. Por tanto, la arquitectura de tres niveles puede facilitar el logro de la verdadera independencia con respecto a los datos, tanto física como lógica. Sin embargo, los dos niveles de correspondencia implican un gasto extra durante la compilación o la ejecución de una consulta o de un programa, lo cual reduce la eficiencia del SGBD. Por ello, son pocos los SGBD que han implementado la arquitectura de tres esquemas completa.

2.3 Lenguajes e interfaces de bases de datos

En la sección 1.4 hemos visto cuáles son los diversos tipos de usuarios que emplean un SGBD. Éste debe ofrecer lenguajes e interfaces apropiadas para cada categoría de usuarios. En esta sección examinaremos los tipos de lenguajes e interfaces que ofrecen los SGBD, así como las categorías de usuarios a las que va dirigida cada interfaz.

2.3.1 *Lenguajes del SGBD*

Una vez que se ha completado el diseño de una base de datos y se ha elegido un SGBD para su implementación, el primer paso será especificar los esquemas conceptual e interno de la base de datos y cualesquiera correspondencias entre ambos. En muchos SGBD en los que no se mantiene una separación estricta de niveles, el DBA y los diseñadores de la base de datos utilizan un mismo lenguaje, el **lenguaje de definición de datos** (DDL: *data definition language*), para definir ambos esquemas. El SGBD contará con un compilador de DDL cuya función será procesar enunciados escritos en el DDL para identificar las descripciones de los elementos de los esquemas y almacenar la descripción del esquema en el catálogo del SGBD.

Cuando en los SGBD se mantenga una clara separación entre los niveles conceptual e interno, el DDL servirá solamente para especificar el esquema conceptual. Se utiliza otro lenguaje, el **lenguaje de definición de almacenamiento** (SDL: *storage definition language*), para especificar el esquema interno. Las correspondencias entre los dos esquemas se pueden especificar en cualquiera de los dos lenguajes. Para una verdadera arquitectura de tres esquemas, necesitaríamos un tercer lenguaje, el **lenguaje de definición de vistas** (VDL: *view definition language*), para especificar las vistas del usuario y sus correspondencias con el esquema conceptual. Una vez que se han compilado los esquemas de la base de datos y que en ésta se han introducido datos, los usuarios requerirán algún mecanismo para manipularla. Las operaciones de manipulación más comunes son la obtención, la inserción, la eliminación y la modificación de los datos. El SGBD ofrece un **lenguaje de manipulación de datos** (DML: *data manipulation language*) para estos fines.

En los SGBD actuales no se acostumbra distinguir entre los tipos de lenguajes mencionados; más bien, se utiliza un amplio lenguaje integrado que cuenta con elementos para definir esquemas conceptuales, definir vistas, manipular datos y definir su almacenamiento. Un ejemplo representativo es el lenguaje de bases de datos relacionales SQL (véase el Cap. 7), que representa una combinación de DDL, VDL, DML y SDL, aunque el componente de SDL está siendo retirado del lenguaje.

Son dos los principales tipos de DML. Los DML **de alto nivel** o **no por procedimientos** se pueden utilizar de manera independiente para especificar operaciones complejas de base de datos en forma concisa. En muchos SGBD es posible introducir interactivamente instrucciones de DML de alto nivel desde una terminal o bien incorporados en un lenguaje de programación de propósito general. En el segundo caso, es preciso identificar los enunciados de DML dentro del programa para que el SGBD pueda procesarlos. Los DML **de bajo nivel** o **por procedimientos** *deben* estar incorporados en un lenguaje de programación de propósito general. Por lo regular, este tipo de DML obtiene registros individuales de la base de datos y los procesa por separado; por tanto, necesita utilizar elementos del lenguaje de programación, como la creación de ciclos, para obtener y procesar cada registro individual de un conjunto de registros. Por esta razón, los DML de bajo nivel se conocen también como DML **de registro por registro**. Los DML de alto nivel, como SQL, pueden especificar y recuperar muchos registros con una sola instrucción de DML, y es por ello que se les llama DML **de conjunto por conjunto** u **orientados a conjuntos**. Las consultas en los DML de alto nivel suelen especificar *qué* datos hay que obtener, y no *cómo* obtenerlos; por ello, tales lenguajes se denominan también **declarativos**.

Siempre que las órdenes de un DML, sean de alto o de bajo nivel, se incorporen en un lenguaje de programación de propósito general, a ese lenguaje se le llamará **lenguaje anfitrión**, y al DML, **sublenguaje de datos**. En los SGBD más recientes, como los sistemas orientados a objetos, el lenguaje anfitrión y el sublenguaje de datos suelen formar un solo lenguaje integrado, como C++. Por otro lado, los DML de alto nivel empleados de manera interactiva e independiente se denominan **lenguajes de consulta**. En general, las órdenes tanto de obtención como de actualización de datos de un DML de alto nivel se pueden utilizar interactivamente, así que se consideran parte del lenguaje de consulta.[†]

Por lo regular, los usuarios finales esporádicos emplean un lenguaje de consulta de alto nivel para especificar sus solicitudes, en tanto que los programadores utilizan el DML en su forma incorporada. Para los usuarios simples y paramétricos casi siempre se incluyen **interfaces amables con el usuario** que permiten interactuar con la base de datos; éstas también pueden aprovecharlas los usuarios esporádicos y otros que no deseen aprender los detalles de un lenguaje de consulta de alto nivel. En seguida analizaremos estos tipos de interfaces.

2.3.2 *Interfaces de SGBD*

Entre las interfaces amables con el usuario que pueden ofrecer los SGBD están las siguientes:

Interfaces basadas en menús. Estas interfaces presentan al usuario listas de opciones, llamadas **menús**, que guían al usuario para formular solicitudes. Los menús hacen innecesario memorizar las órdenes y la sintaxis específicas de un lenguaje de consulta, pues permiten

[†]En vista del significado de la palabra *consulta* en español, sólo debería usarse en realidad para describir la obtención de datos, no la actualización.

construir la solicitud paso por paso eligiendo las opciones de los menús que el sistema presenta. Los menús desplegables son una técnica cada vez más socorrida en las interfaces del usuario basadas en ventanas, y a menudo se utilizan en las **interfaces para hojear**, que permiten al usuario examinar el contenido de una base de datos en una forma no estructurada.

Interfaces gráficas. Las interfaces gráficas suelen presentar al usuario los esquemas en forma de diagrama, y éste puede entonces especificar una consulta manipulando el diagrama. En muchos casos, las interfaces gráficas se combinan con menús. Casi todas estas interfaces se valen de un **dispositivo apuntador**, como el ratón (*mouse*), para escoger ciertas partes del diagrama de esquema que se exhibe.

Interfaces basadas en formas. Las interfaces basadas en formas presentan una **forma** a cada usuario. Éste puede entonces llenar todos los espacios de la forma para insertar datos nuevos, o bien llenar sólo ciertos espacios, en cuyo caso el SGBD obtendrá los registros que coincidan con los datos especificados. Las formas suelen diseñarse y programarse para los usuarios simples como interfaces de transacciones programadas. Muchos SGBD cuentan con lenguajes especiales, los *lenguajes de especificación de formas*, con los que los programadores pueden especificar dichas formas. Algunos sistemas cuentan con utilerías que definen formas al permitir que el usuario construya interactivamente una forma de muestra en la pantalla.

Interfaces de lenguaje natural. Estas interfaces aceptan solicitudes escritas en inglés o en algún otro idioma e intentan "entenderlas". Las interfaces de lenguaje natural suelen tener su propio "esquema", similar al esquema conceptual de la base de datos. La interfaz consulta las palabras de su esquema, y también un conjunto de palabras estándar, para interpretar la solicitud. Si la interpretación tiene éxito, la interfaz genera una consulta de alto nivel que corresponde a la solicitud en lenguaje natural y la envía al SGBD para su procesamiento; en caso contrario, se inicia un diálogo con el usuario para esclarecer la solicitud.

Interfaces para usuarios paramétricos. Los usuarios paramétricos, como los cajeros de un banco, a menudo tienen un conjunto pequeño de operaciones que deben realizar repetidamente. Los analistas de sistemas y los programadores diseñan e implementan una interfaz especial para una clase conocida de usuarios simples. Casi siempre se incluye un conjunto reducido de órdenes abreviadas, con el fin de reducir al mínimo el número de digitaciones requeridas para cada solicitud. Por ejemplo, se pueden programar las teclas de funciones de una terminal para que inicien las diversas órdenes. Con esto el usuario paramétrico puede trabajar con el menor número de digitaciones posible.

Interfaces para el DBA. En su mayoría, los sistemas de base de datos contienen órdenes privilegiadas que sólo el personal del DBA puede utilizar. Entre ellas están las órdenes para crear cuentas, establecer los parámetros del sistema, otorgar autorizaciones a las cuentas, modificar los esquemas y reorganizar la estructura de almacenamiento de una base de datos.

2.4 El entorno de un sistema de base de datos*

Los SGBD son sistemas de software muy complejos. En esta sección estudiaremos los tipos de componentes del software que constituyen un SGBD y los tipos de software del sistema de cómputo con los cuales interactúa el SGBD.

2.4.1 *Módulos componentes de un SGBD*

La figura 2.3 ilustra los componentes de un SGBD representativo. La base de datos y el catálogo del SGBD casi siempre se almacenan en disco. El acceso al disco suele controlarlo primordialmente el *sistema operativo* (OS: *operating system*), que programa la entrada/salida del disco. Un módulo **gestor de datos almacenados** del SGBD, de más alto nivel, controla el acceso a la información del SGBD almacenada en el disco, ya sea que forme parte de la base de datos o del catálogo. Las líneas punteadas y los círculos rotulados A, B, C, D y E en la figura 2.3 ilustran accesos que están bajo el control de este gestor de datos almacenados. Este último puede emplear servicios básicos del OS para transferir los datos de bajo nivel entre el disco y la memoria principal del computador, pero controla otros aspectos de la transferencia de datos, como el manejo de las áreas de almacenamiento intermedio (*buffers*) en la memoria principal. Una vez que los datos estén en dicho almacenamiento intermedio, otros módulos del SGBD podrán procesarlos.

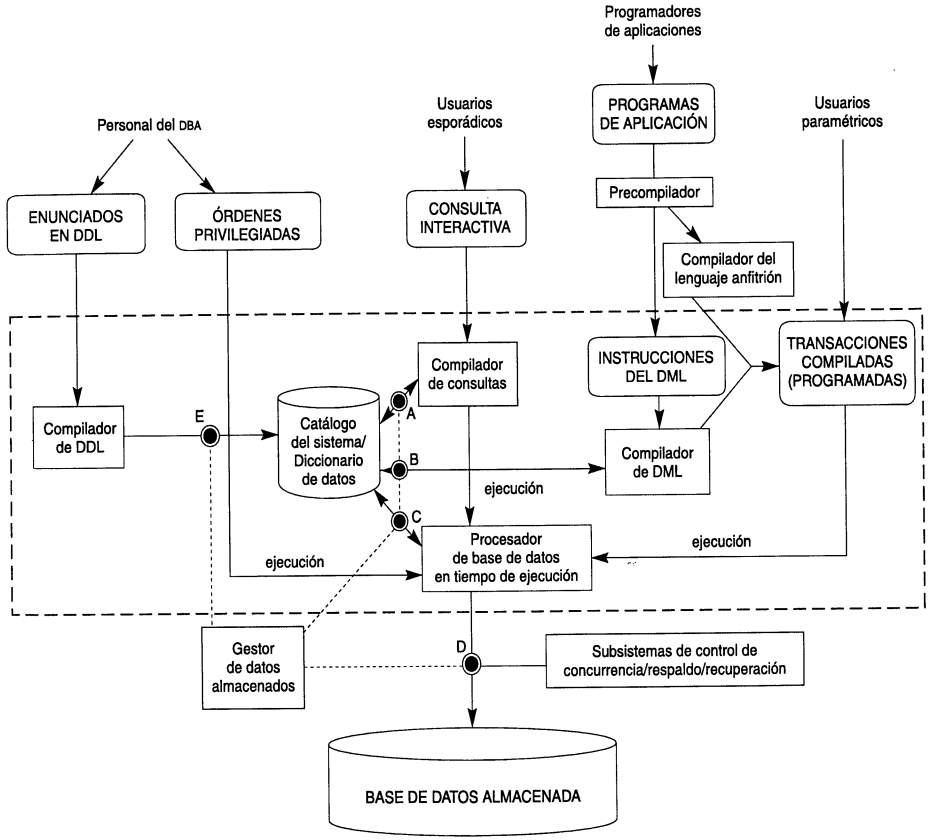


Figura 2.3 Componentes de un SGBD. Las líneas punteadas indican accesos que están bajo el control del gestor de datos almacenados.

El **compilador de DDL** procesa las definiciones de esquemas, especificadas en el DDL, y almacena las descripciones de los esquemas (metadatos) en el catálogo del SGBD. El catálogo contiene información como los nombres de los archivos y de los elementos de información, los detalles de almacenamiento de cada archivo, la información de correspondencia entre los esquemas y las restricciones. Los módulos del SGBD que necesiten conocer esta información deberán consultar el catálogo.

El **procesador de la base de datos en tiempo de ejecución** se encarga de los accesos a ella durante la ejecución; recibe operaciones de obtención o de actualización y las ejecuta sobre la base de datos. El acceso al disco se tiene mediante el gestor de datos almacenados. El **compilador de consultas** maneja las consultas de alto nivel que se introducen interactivamente. Analiza la sintaxis y el contenido de las consultas y luego genera llamadas al procesador en tiempo de ejecución para atender la solicitud.

El **precompilador** extrae órdenes en DML de un programa de aplicación escrito en un lenguaje de programación anfitrión. Estas órdenes se envían al **compilador de DML** para convertirlas en código objeto para el acceso a la base de datos, y el resto del programa se envía al compilador del lenguaje anfitrión. El código objeto de las órdenes en DML y el del resto del programa se enlazan, formando una transacción programada cuyo código ejecutable incluye llamadas al procesador de la base de datos durante el tiempo de ejecución.

Con la figura 2.3 no se intenta describir un SGBD específico; más bien, lo utilizamos para ilustrar los módulos representativos de un SGBD. El SGBD interactúa con el sistema operativo cuando se requiere acceso al disco (a la base de datos o al catálogo). Si muchos usuarios comparten el mismo sistema de cómputo, el OS programará las solicitudes de acceso a disco del SGBD junto con otros procesos. El SGBD también se comunica con los compiladores de los lenguajes de programación anfitriones, y puede ofrecer interfaces amables con el usuario como ayuda para cualesquiera de los tipos de usuarios mostrados en la figura 2.3 cuando especifican sus solicitudes.

2.4.2 *Utilerías del sistema de base de datos*

Además de los módulos de software que acabamos de describir, casi todos los SGBD cuentan con **utilerías de base de datos** que ayudan al DBA a manejar el sistema. Las utilerías comunes efectúan los siguientes tipos de funciones:

Carga: Se utiliza una utilería de carga para cargar archivos de datos ya existentes —como archivos de texto o secuenciales— en la base de datos. Por lo regular se especifican a la utilería el formato actual (fuente) de los datos y la estructura de archivos de la base de datos deseada (destino). En seguida, la utilería modifica automáticamente el formato de los datos y los almacena en la base de datos. Con la proliferación de los SGBD, la transferencia de datos de un SGBD a otro se ha vuelto cosa común en muchas organizaciones. Algunos proveedores ahora ofrecen productos que generan los programas de carga apropiados, de acuerdo con las descripciones de almacenamiento de las bases de datos fuente y destino (esquemas internos). Un ejemplo es el sistema EXTRACT de Evolutionary Technologies. Estos paquetes se denominan también **herramientas de conversión**.

Respaldo: Las utilerías de respaldo crean una copia de seguridad de la base de datos, casi siempre virtiendo toda la base de datos en cinta. La copia de seguridad puede servir para restaurar la base de datos en caso de un fallo catastrófico.

Reorganización de archivos: Esta utilería puede servir para modificar la organización de los archivos de la base de datos con el fin de mejorar el rendimiento.

Vigilancia del rendimiento: Las utilerías de este tipo vigilan la utilización de la base de datos y proporcionan datos estadísticos al DBA, el cual los utiliza para decidir, por ejemplo, si conviene reorganizar los archivos a fin de mejorar el rendimiento.

Puede haber otras utilerías para ordenar archivos, comprimir datos, vigilar los accesos de los usuarios y efectuar otras funciones. Otra utilería que puede resultar muy provechosa en las organizaciones grandes es un **sistema de diccionario de datos** expandido. Además de almacenar información del catálogo relativa a los esquemas y las restricciones, el diccionario de datos guarda información de otra índole, como decisiones de diseño, normas de utilización, descripciones de los programas de aplicación e información sobre los usuarios. Los usuarios o el DBA pueden tener acceso a esta información en caso de necesitarla. Las utilerías del diccionario de datos son similares al catálogo del SGBD, pero contienen información más amplia y variada, y las utilizan principalmente los usuarios, no el software del SGBD. La combinación de catálogo/diccionario de datos, a la que tienen acceso tanto los usuarios como el SGBD, se denomina **directorio de datos** o diccionario de datos **activo**. Si un diccionario de datos es accesible para los usuarios y para el DBA, pero no para el software del SGBD, se le llama **pasivo**.

2.4.3 *Recursos de comunicaciones*

El SGBD también debe interactuar con **software de comunicaciones**, cuya función es permitir que los usuarios situados en lugares remotos respecto al sistema de base de datos tengan acceso a éste a través de terminales de computador, estaciones de trabajo o sus microcomputadores o minicomputadores locales. Éstos se conectan al sitio de la base de datos por medio de equipos de comunicación de datos: líneas telefónicas, redes de larga distancia o dispositivos de comunicación por satélite. Muchos sistemas comerciales de bases de datos tienen paquetes de comunicaciones que funcionan con el SGBD. El sistema integrado de SGBD y comunicación de datos se denomina sistema **DB/DC** (*database/data communications*).

Por añadidura, algunos SGBD distribuidos están físicamente dispersos en varias máquinas. En este caso, se requieren redes de comunicaciones para conectar las máquinas. Con frecuencia se trata de *redes de área local* (LAN: *local area networks*), pero también pueden ser de otro tipo. El término **arquitectura cliente-servidor** se usa para caracterizar un SGBD cuando la aplicación se ejecuta físicamente en una máquina, llamada **cliente**, y otra, el **servidor**, se encarga del almacenamiento y el acceso a los datos. Los proveedores ofrecen diversas combinaciones de clientes y servidores; por ejemplo, un servidor para varios clientes.

2.5 Clasificación de los sistemas de gestión de bases de datos*

El principal criterio que suele utilizarse para clasificar los SGBD es el **modelo de datos** en que se basan. Los modelos de datos empleados con mayor frecuencia en los SGBD comerciales actuales son el relacional, el de red y el jerárquico. Algunos SGBD recientes se basan en modelos

orientados a objetos o conceptuales. Clasificaremos los SGBD como **relacionales, de red, jerárquicos, orientados a objetos y otros.**

Un segundo criterio para clasificar los SGBD es el **número de usuarios** a los que da servicio el sistema. Los sistemas **monousuario** sólo atienden a un usuario a la vez, y su principal uso se da en los computadores personales. Los sistemas **multiusuario**, entre los que se cuentan la mayor parte de los SGBD, atienden a varios usuarios al mismo tiempo.

Un tercer criterio es el **número de sitios** en los que está distribuida la base de datos. Casi todos los SGBD son **centralizados**; esto es, sus datos se almacenan en un solo computador. Los SGBD centralizados pueden atender a varios usuarios, pero el SGBD y la base de datos en sí residen por completo en un solo computador. En los **SGBD distribuidos (SGBDD)** la base de datos real y el software del SGBD pueden estar distribuidos en varios sitios, conectados por una red de computadores. Los **SGBDD homogéneos** utilizan el mismo software de SGBD en múltiples sitios. Una tendencia reciente consiste en crear software para tener acceso a varias bases de datos autónomas preexistentes almacenadas en SGBD **heterogéneos**. Esto da lugar a los **SGBD federados** (o **sistemas multibase de datos**) en los que los SGBD participantes están débilmente acoplados y tienen cierto grado de autonomía local. Muchos SGBDD emplean una arquitectura cliente-servidor. Hablaremos de los SGBDD y de la arquitectura cliente-servidor en el capítulo 23.

Un cuarto criterio es el **costo** del SGBD. La mayor parte de los paquetes de SGBD cuestan entre 10 000 y 100 000 dólares estadounidenses. Los sistemas monousuario más económicos para microcomputadores tienen un costo de entre $100 y $3000. En el otro extremo, unos cuantos paquetes muy completos cuestan más de $100 000.

También podemos clasificar los SGBD con base en los **tipos de camino de acceso** de que disponen para almacenar los archivos. Una familia muy conocida de SGBD se basa en estructuras de archivos invertidos. Por último, los SGBD pueden ser **de propósito general** o **de propósito especial**. Cuando el rendimiento es de primordial importancia, se puede diseñar y construir un SGBD de propósito especial para una aplicación específica, y este sistema no servirá para otras aplicaciones. Muchos sistemas de reservaciones de líneas aéreas y de directorio telefónico son SGBD de propósito especial, y pertenecen a la categoría de **sistemas de procesamiento de transacciones en línea** (OLTP: *on-line transaction processing*), que deben atender un gran número de transacciones concurrentes sin imponer retrasos excesivos.

Analicemos brevemente el criterio principal con que se clasifican los SGBD: el modelo de datos. El modelo de datos **relacional** representa una base de datos como una colección de tablas, cada una de las cuales se puede almacenar en forma de archivo individual. La base de datos de la figura 1.2 se mostró de manera muy similar a una representación relacional. Casi todas las bases de datos relacionales tienen lenguajes de consulta de alto nivel y manejan una forma limitada de vistas de usuario. En los capítulos 6 al 9 analizaremos el modelo relacional, sus lenguajes y operaciones, así como un ejemplo de sistema.

El modelo de datos **de red** representa los datos como tipos de registros y también representa un tipo limitado de vínculos 1:*N*, llamado tipo de conjunto. La figura 2.4 muestra en forma de diagrama un esquema de red para la base de datos de la figura 1.2, donde los tipos de registros aparecen como rectángulos y los tipos de conjuntos como flechas dirigidas rotuladas. El modelo de red, también conocido como modelo CODASYL DBTG,[†] tiene un lenguaje de registro por

[†]CODASYL DBTG significa Conference on Data Systems Language Data Base Task Group (grupo de trabajo sobre bases de datos de la conferencia sobre lenguajes de sistemas de datos), que es el comité que especificó el modelo de red y su lenguaje.

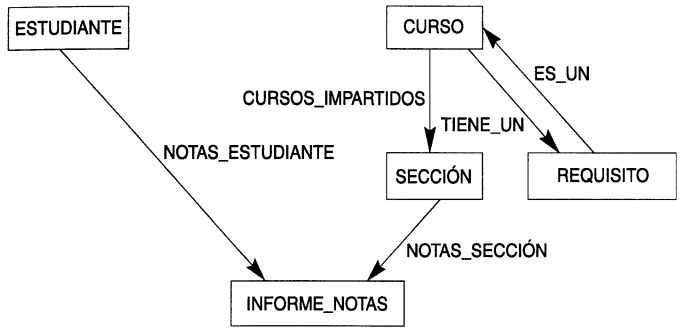


Figura 2.4 Esquema de red.

registro asociado que se debe incorporar en un lenguaje de programación anfitrión. Presentaremos el modelo de red y su lenguaje en el capítulo 10.

El modelo **jerárquico** representa los datos como estructuras jerárquicas de árbol. Cada jerarquía representa varios registros relacionados entre sí. No existe un lenguaje estándar para el modelo jerárquico, aunque la mayor parte de los SGBD jerárquicos cuentan con lenguajes de registro por registro. Trataremos el modelo jerárquico en el capítulo 11.

El modelo **orientado a objetos** define una base de datos en términos de objetos, sus propiedades y sus operaciones. Los objetos con la misma estructura y comportamiento pertenecen a una **clase**, y las clases se organizan en jerarquías o grafos acíclicos. Las operaciones de cada clase se especifican en términos de procedimientos predefinidos llamados **métodos**. Ya hay ahora en el mercado varios sistemas basados en el paradigma orientado a objetos. Además, los SGBD relacionales han estado extendiendo sus modelos para incorporar conceptos orientados a objetos y otras capacidades; a éstos se les conoce como **sistemas relacionales extendidos**. Analizaremos con detalle los modelos orientados a objetos en el capítulo 22.

En el siguiente capítulo presentaremos los conceptos del modelo de **entidad-vínculo**, un modelo de datos conceptual de alto nivel que ha adquirido gran popularidad en el diseño de bases de datos.

2.6 Resumen

En este capítulo presentamos los conceptos más importantes empleados en los sistemas de bases de datos. Definimos qué es un modelo de datos y distinguimos tres categorías principales de modelos de datos:

- Modelos de datos de alto nivel o conceptuales (entidad—vínculo).
- Modelos de datos de implementación (basados en registros, orientados a objetos).
- Modelos de datos de bajo nivel, o físicos.

Establecimos la diferencia entre el esquema, o descripción de una base de datos, de la base de datos en sí misma. El esquema no cambia muy a menudo, en tanto que la base de datos cambia cada vez que se insertan, eliminan o modifican datos. En seguida describimos la arquitectura de SGBD de tres esquemas, que contempla tres niveles de esquemas:

- Los esquemas externos, que describen las visitas de diferentes grupos de usuarios.
- El esquema conceptual, que es una descripción de alto nivel de toda la base de datos.
- El esquema interno, que describe las estructuras de almacenamiento de la base de datos.

Todo SGBD que separe claramente los tres niveles deberá tener correspondencias entre los esquemas para transformar las solicitudes y resultados de un nivel al siguiente. La mayoría de los SGBD no separan los tres niveles por completo. Utilizamos la arquitectura de tres esquemas para definir los conceptos de independencia lógica y física con respecto a los datos.

En la sección 2.3 estudiamos los principales tipos de lenguajes que manejan los SGBD. Los lenguajes de definición de datos (DDL) sirven para definir el esquema conceptual de la base de datos. En casi todos los SGBD, el DDL define también las vistas de usuario y las estructuras de almacenamiento; en otros SGBD hay lenguajes individuales (VDL, SDL) para especificar vistas y estructuras de almacenamiento. El SGBD compila todas las definiciones de esquemas y las almacena en el catálogo del SGBD. Los lenguajes de manipulación de datos (DML) sirven para especificar operaciones de obtención y actualización de datos. Los DML pueden ser de alto nivel (no por procedimientos) o de bajo nivel (por procedimientos). Un DML de alto nivel puede estar incorporado en un lenguaje de programación anfitrión, o emplearse como lenguaje independiente; en el segundo caso suele recibir el nombre de lenguaje de consulta.

Analizamos los diferentes tipos de interfaces que ofrecen los SGBD, y los tipos de usuarios con los que está asociada cada interfaz. En seguida hablamos del entorno del sistema de base de datos, los componentes usuales del software del SGBD y las utilerías del SGBD que ayudan a los usuarios y al DBA a llevar a cabo sus tareas.

Por último, clasificamos los SGBD de acuerdo con el modelo de datos, el número de usuarios, el número de sitios, el costo, el tipo de caminos de acceso y la generalidad. La clasificación más importante de los SGBD se basa en el modelo de datos, e hicimos una breve explicación de los principales modelos que se utilizan en los SGBD comerciales del presente.

Preguntas de repaso

2.1. Defina los siguientes términos: *modelo de datos, esquema de base de datos, estado de base de datos, esquema interno, esquema conceptual, esquema externo, independencia con respecto a los datos,* DDL, DML, SDL, VDL, *lenguaje de consulta, lenguaje anfitrión, sublenguaje de datos, utilería de bases de datos, diccionario de datos activo, diccionario de datos pasivo, catálogo, arquitectura cliente-servidor.*

2.2. Analice las principales categorías de modelos de datos.

2.3. ¿Qué diferencia hay entre un esquema de base de datos y un estado de base de datos?

2.4. Describa la arquitectura de tres esquemas. ¿Por qué son necesarias las correspondencias entre los distintos niveles de esquemas? ¿De qué manera apoyan esta arquitectura los diferentes lenguajes de definición de esquemas?

2.5. ¿Qué diferencia hay entre la independencia lógica y la independencia física con respecto a los datos? ¿Cúal es más fácil de lograr? ¿Por qué?

2.6. ¿Qué diferencia hay entre los DML por procedimientos y no por procedimientos?

2.7. Analice los distintos tipos de interfaces amables con el usuario y los tipos de usuarios que suelen utilizarlas.

2.8. ¿Con qué otro software del computador interactúa un SGBD?

2.9. Analice algunos tipos de utilerías de base de datos y sus funciones.

Ejercicios

2.10. Sugiera diferentes usuarios para la base de datos de la figura 1.2. ¿Qué tipos de aplicaciones necesitaría cada usuario? ¿A qué categoría de usuario pertenecería cada uno, y qué tipo de interfaz necesitaría?

2.11. Escoja una aplicación de base de datos que conozca bien. Diseñe un esquema y prepare una base de datos de muestra para esa aplicación, utilizando la notación de las figuras 2.1 y 1.2. ¿Qué tipos de información y restricciones adicionales querría representar en el esquema? Piense en varios usuarios para su base de datos, y diseñe una vista para cada uno.

Bibliografía selecta

Muchos textos sobre bases de datos analizan los diversos conceptos que presentamos aquí, entre ellos los de Wiederhold (1986), Ullman (1988), Date (1990) y Korth y Silberschatz (1991). Tsichritzis y Lochovsky (1982) es un libro de texto sobre modelos de datos. Tsichritzis y Klug (1978) y Jardine (1977) presentan la arquitectura de tres esquemas, sugerida inicialmente en el informe CODASYL DBTG (1971) y más adelante en un informe del American National Standards Institute (ANSI) (1975).

CAPÍTULO 3

Modelado de datos con el enfoque entidad-vínculo

El objetivo de este capítulo es presentar los conceptos del modelo de **entidad-vínculo**, o de **entidad-relación** (ER: *Entity-Relationship*), que es un modelo de datos conceptual de alto nivel muy utilizado. Este modelo y sus variaciones se emplean a menudo en el diseño conceptual de aplicaciones de bases de datos, y muchas herramientas de diseño de bases de datos aplican sus conceptos. Describiremos los conceptos básicos de estructuración de datos y las restricciones del modelo ER, y estudiaremos su empleo en el diseño de esquemas conceptuales para aplicaciones de base de datos.

Este capítulo está organizado de la siguiente manera. En la sección 3.1 analizaremos el papel de los modelos de datos conceptuales en el diseño de bases de datos. En la sección 3.2 estudiaremos los requerimientos para un ejemplo de aplicación de base de datos, a fin de ilustrar el empleo de los conceptos del modelo ER. Esta base de datos de ejemplo se usará también en capítulos subsecuentes. En la sección 3.3 presentaremos los conceptos del modelo ER, e introduciremos gradualmente la técnica de diagramación para representar un esquema ER. La sección 3.4 es un repaso de la notación para los diagramas ER, y en la sección 3.5 se analizará el problema de cómo elegir los nombres para los elementos de los esquemas de base de datos. En la sección 3.6 trataremos los vínculos ternarios y de grados más altos. Por último, concluiremos con un resumen en la sección 3.7.

El lector puede pasar por alto la sección 3.6 y examinar superficialmente parte del material con que se cierra la sección 3.3 si así lo desea. Por otro lado, si prefiere una cobertura más completa de los conceptos de modelado de datos y del diseño conceptual de bases de datos, puede continuar con el material del capítulo 21 después de terminar con el 3. En el capítulo 21 describiremos las extensiones del modelo ER que llevan al modelo ER extendido (EER: *Enhanced* ER), incluidas la especialización y la generalización, técnicas para especificar transacciones de alto nivel, y un análisis de las restricciones de integridad.

3.1 Modelos de datos conceptuales de alto nivel para diseño de bases de datos

La figura 3.1 muestra una descripción simplificada del proceso de diseño de bases de datos. El primer paso que aparece es la **recolección y análisis de requerimientos**, durante la cual los diseñadores entrevistan a los futuros usuarios de la base de datos para entender y documentar sus requerimientos de información. El resultado de este paso será un conjunto de requerimientos del usuario redactado en forma concisa. Estos requerimientos deben especificarse en la forma más detallada y completa que sea posible. En paralelo con la especificación de los requerimentos de datos, conviene especificar los *requerimientos funcionales* conocidos de la aplicación. Éstos consisten en las operaciones definidas por el usuario (o **transacciones**) que se aplicarán a la base de datos, e incluyen la obtención de datos y la actualización. Se acostumbra usar técnicas como los *diagramas de flujos de datos* para especificar los requerimientos funcionales.

Una vez recabados y analizados todos los requerimientos, el siguiente paso es crear un **esquema conceptual** para la base de datos mediante un modelo de datos conceptual de alto nivel. Este paso se denomina **diseño conceptual de la base de datos**. El esquema conceptual es una descripción concisa de los requerimientos de información de los usuarios, y contiene descripciones detalladas de los tipos de datos, los vínculos y las restricciones; éstas se expresan mediante los conceptos del modelo de datos de alto nivel. Puesto que estos conceptos no incluyen detalles de la implementación, suelen ser más fáciles de entender, de modo que pueden servir para comunicarse con usuarios no técnicos. El esquema conceptual de alto nivel también puede servir como referencia para asegurarse de satisfacer todos los requerimientos de los usuarios y de que no haya conflictos entre dichos requerimientos. Este enfoque permite a los diseñadores de la base de datos concentrarse en especificar las propiedades de los datos, sin preocuparse por detalles del almacenamiento; en consecuencia, tienen menos problemas para elaborar un buen diseño conceptual.

Una vez diseñado el esquema conceptual, es posible utilizar las operaciones básicas del modelo de datos para especificar transacciones de alto nivel que correspondan a las operaciones definidas por el usuario que se hayan identificado durante el análisis funcional. Esto también sirve para confirmar que el esquema conceptual satisfaga todos los requerimientos funcionales identificados. Se puede modificar en este momento el esquema conceptual si no resulta factible especificar algunos requerimientos funcionales en el esquema inicial.

El siguiente paso en este proceso de diseño consiste en implementar de hecho la base de datos con un SGBD comercial. La mayoría de los SGBD disponibles hoy en el mercado utilizan un modelo de datos de implementación, así que el esquema conceptual se traduce del modelo de datos de alto nivel al modelo de datos de implementación. Este paso se denomina **diseño lógico de la base de datos** o **transformación de modelos de datos**, y su resultado es un esquema de base de datos especificado en el modelo de datos de implementación del SGBD.

El paso final es la fase de **diseño físico de la base de datos**, durante la cual se especifican las estructuras de almacenamiento internas y la organización de los archivos de la base de datos. En paralelo con estas actividades, se diseñan e implementan programas de aplicación en forma de transacciones de base de datos que correspondan a las especificaciones de transacciones de alto nivel. Trataremos el proceso de diseño de bases de datos con mayor detalle en el capítulo 14.

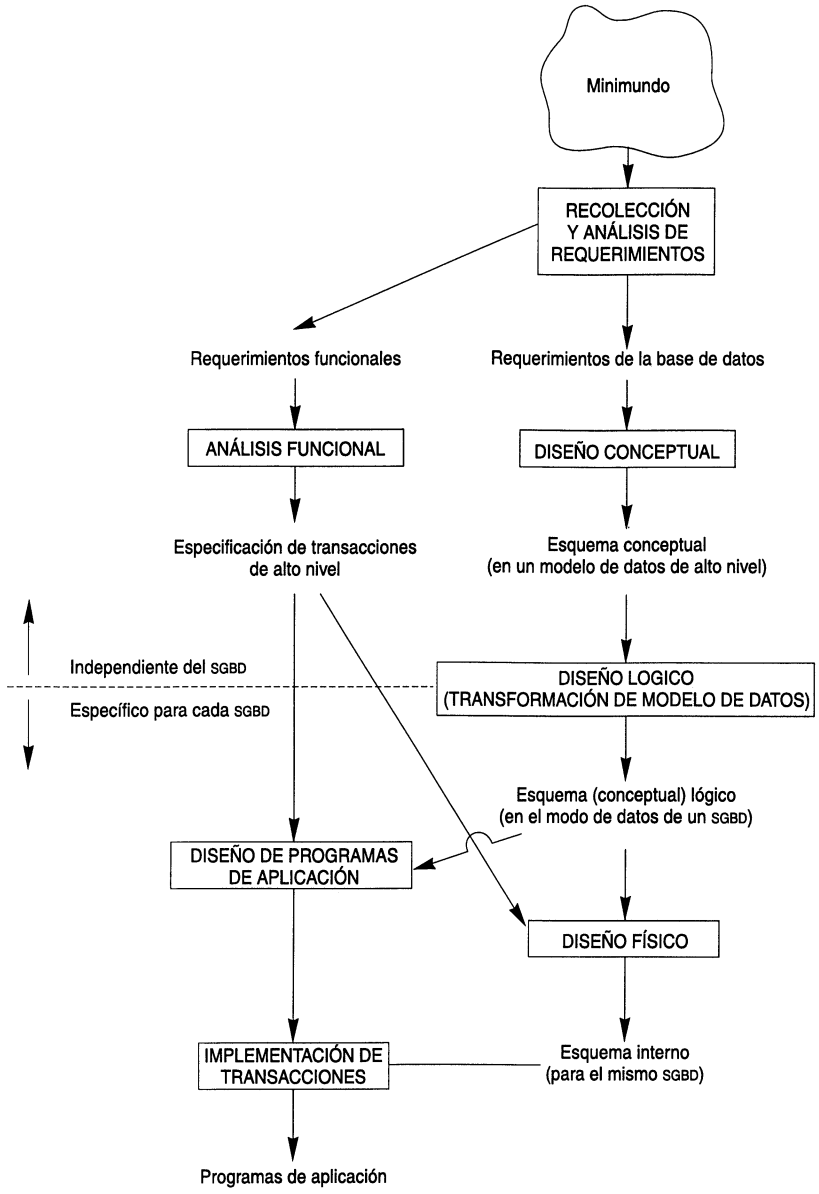


Figura 3.1 Fases del diseño de bases de datos (simplificado).

En este capítulo sólo presentaremos los conceptos del modelo ER que intervienen en el diseño de esquemas conceptuales. Las operaciones de este modelo, que pueden servir para especificar transacciones definidas por el usuario, se verán en el capítulo 21.

3.2 Un ejemplo

En esta sección describiremos una base de datos de ejemplo, llamada COMPAÑÍA, que servirá para ilustrar los conceptos del modelo ER y su uso en el diseño de esquemas. Aquí vamos a mencionar los requerimientos de información de esta base de datos, y en seguida crearemos su esquema conceptual paso por paso al tiempo que presentamos los conceptos de modelado que intervienen en el modelo ER. La base de datos COMPAÑÍA se ocupa de los empleados, departamentos y proyectos de una empresa. Vamos a suponer que, una vez concluida la fase de recolección y análisis de requerimientos, los diseñadores de la base de datos redactaron la siguiente descripción del "minimundo" (la parte de la compañía que se representará en la base de datos):

1. La compañía está organizada en departamentos. Cada departamento tiene un nombre único, un número único y un cierto empleado que lo dirige, y nos interesa la fecha en que dicho empleado comenzó a dirigir el departamento. Un departamento puede estar distribuido en varios lugares.
2. Cada departamento controla un cierto número de proyectos, cada uno de los cuales tiene un nombre y un número únicos, y se efectúa en un solo lugar.
3. Almacenaremos el nombre, número de seguro social, dirección, salario, sexo y fecha de nacimiento de cada empleado. Todo empleado está asignado a un departamento, pero puede trabajar en varios proyectos, que no necesariamente estarán controlados por el mismo departamento. Nos interesa el número de horas por semana que un empleado trabaja en cada proyecto, y también quién es el supervisor de cada empleado.
4. Queremos mantenernos al tanto de los dependientes de cada empleado con el fin de administrar los términos de sus seguros. Almacenaremos el nombre, sexo y fecha de nacimiento de cada dependiente, y su parentesco con el empleado.

La figura 3.2 muestra cómo se puede presentar el esquema de esta aplicación de base de datos mediante la notación gráfica conocida como **diagramas ER**. En la siguiente sección describiremos el proceso de derivar este esquema a partir de los requerimientos declarados —y explicaremos la notación de los diagramas ER— conforme vayamos presentando los conceptos del modelo ER.

3.3 Conceptos del modelo ER

El modelo ER describe los datos como entidades, vínculos y atributos. En la sección 3.3.1 estudiaremos los conceptos de entidades y sus atributos; en la sección 3.3.2 analizaremos los tipos de entidades y los atributos clave; en la sección 3.3.3 veremos los tipos de vínculos y

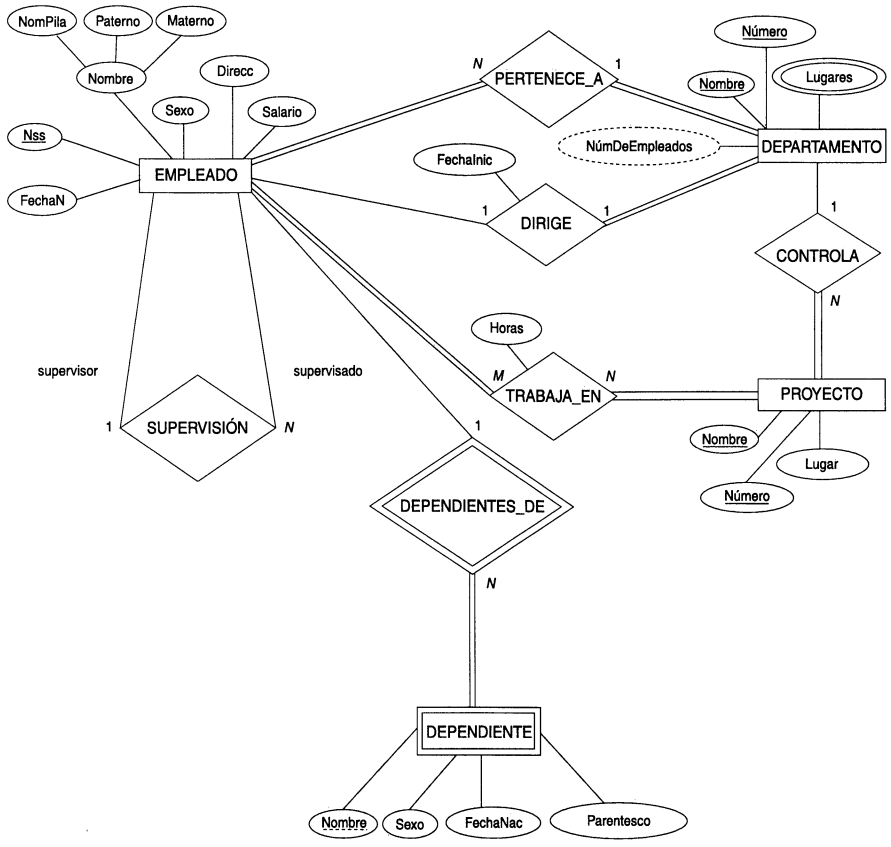


Figura 3.2 Diagrama de esquema ER para la base de datos COMPAÑÍA.

sus restricciones estructurales; en la sección 3.3.4 hablaremos de los tipos de entidades débiles, y en la sección 3.3.5 ilustraremos cómo se usan los conceptos del ER en el diseño de la base de datos COMPAÑÍA.

3.3.1 *Entidades y atributos*

El objeto básico que se representa en el modelo ER es la **entidad**: una "cosa" del mundo real con existencia independiente. Una entidad puede ser un objeto con existencia física —una cierta persona, un automóvil, una casa o un empleado— o un objeto con existencia conceptual, como una compañía, un puesto de trabajo o un curso universitario. Cada entidad tiene propiedades específicas, llamadas **atributos**, que la describen. Por ejemplo, una entidad empleado puede describirse por su nombre, su edad, su dirección, su salario y su puesto de trabajo. Una entidad particular tendrá un **valor** para cada uno de sus atributos; los valores de los atributos que describen a cada entidad constituyen una parte decisiva de los datos almacenados en la base de datos.

La figura 3.3 muestra dos entidades y los valores de sus atributos. La entidad empleado e_1 tiene cuatro atributos: Nombre, Dirección, Edad y Teléfono; sus valores son "Juan Sánchez", "Valle 2311, Medellín, Colombia 77001", "55", y "713-749-2630", respectivamente. La entidad compañía c_1 tiene tres atributos: Nombre, Ubicación y Presidente; sus valores son "Lubricol", "Medellín" y "Juan Sánchez", respectivamente.

Tipos de atributos. En el modelo ER se manejan varios tipos distintos de atributos: *simples* o *compuestos*; *monovaluados* o *multivaluados*, y *almacenados* o *derivados*. Primero definiremos estos tipos de atributos e ilustraremos su uso mediante ejemplos; luego presentaremos el concepto de *valor nulo* para un atributo.

Los **atributos compuestos** se pueden dividir en componentes más pequeños, que representan atributos más básicos con su propio significado independiente. Por ejemplo, el atributo Dirección de la entidad empleado que se muestra en la figura 3.3 se puede subdividir en Domicilio, Ciudad, País y CP, con los valores "Valle 2311", "Medellín", "Colombia" y "77001". Los atributos no divisibles se denominan atributos **simples** o **atómicos**. Los atributos compuestos pueden formar una jerarquía; por ejemplo, Domicilio aún se podría subdividir en tres atributos simples, Calle, NúmExterior y NúmInterior, como se aprecia en la figura 3.4. El valor de un atributo compuesto es la concatenación de los valores de los atributos simples que lo constituyen.

Los atributos compuestos son útiles para modelar situaciones en las que un usuario en ocasiones hace referencia al atributo compuesto como una unidad, pero otras veces se refiere específicamente a sus componentes. Si sólo se hace referencia al atributo compuesto como un todo, no hay necesidad de subdividirlo en sus atributos componentes. Por ejemplo, si no hay necesidad de referirse a los componentes individuales de una dirección (CP, Calle, etc.), la dirección completa se designará como atributo simple.

En su mayoría, los atributos tienen un solo valor para una entidad en particular, y reciben el calificativo de **monovaluados**. Por ejemplo, Edad es un atributo monovaluado de Persona. Hay casos en los que un atributo puede tener un conjunto de valores para la misma entidad; por ejemplo, un atributo Colores para un automóvil, o un atributo GradosUniversitarios para una persona. Los coches de un solo color sólo tienen un valor de Colores, pero los de dos tonos pueden tener dos. De manera similar, una persona podría no tener grado universitario alguno, otra podría tener uno, y una tercera podría tener dos o más grados; así, diferentes personas pueden tener distintos *números de valores* para el atributo GradosUniversitarios. Los atributos de este tipo se denominan **multivaluados**, y pueden tener límites inferior y superior del número de valores para una entidad individual. Por ejemplo, el atributo

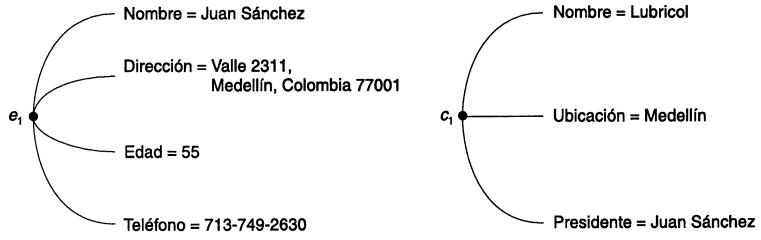


Figura 3.3 Dos entidades y sus valores de atributos.

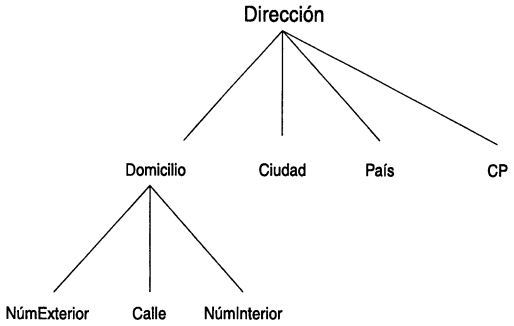


Figura 3.4 Jerarquía de atributos compuestos.

Colores de un automóvil puede tener entre uno y cinco valores, si suponemos que los coches pueden tener como máximo cinco colores.

En algunos casos se relacionan dos (o más) valores de atributos; por ejemplo, los atributos Edad y FechaNacimiento de una persona. Para una entidad persona en particular, el valor de Edad se puede determinar a partir de la fecha actual y el valor de FechaNacimiento de esa persona. Por tanto, se dice que el atributo Edad es un **atributo derivado**, y que es **derivable** del atributo FechaNacimiento, el cual es un **atributo almacenado**. Algunos valores de atributos se pueden derivar de *entidades relacionadas*; por ejemplo, es posible derivar un atributo NúmeroDeEmpleados de una entidad departamento si se cuenta el número de empleados relacionados con (que trabajan en) ese departamento.

En algunos casos, una cierta entidad podría no tener ningún valor aplicable para un atributo. Por ejemplo, el atributo NúmInterior de una dirección sólo se aplica a direcciones que corresponden a edificios de departamentos o a condominios, pero no a otros tipos de residencias unifamiliares. De manera similar, un atributo GradosUniversitarios sólo se aplica a personas con grados universitarios. En situaciones de este tipo se crea un valor especial llamado **nulo**. La dirección de una casa unifamiliar tendría nulo en su atributo NúmInterior, y una persona sin grados universitarios tendría nulo en GradosUniversitarios. También podemos usar nulo si no conocemos el valor de un atributo para una entidad específica; por ejemplo, si no sabemos cuál es el teléfono de "Juan Sánchez" en la figura 3.3. El significado del primer tipo de nulo es *no aplicable*, en tanto que el significado del segundo es *desconocido*. La categoría desconocido del valor nulo puede clasificarse en uno de dos casos. El primero se da cuando se sabe que el valor del atributo existe, pero *falta*; por ejemplo, cuando aparece como nulo el valor del atributo Altura de una persona. El segundo caso ocurre cuando *no se sabe* si el valor del atributo existe; por ejemplo, cuando aparece como nulo el valor del Teléfono de una persona.

3.3.2 *Tipos de entidades, conjuntos de valores y atributos clave*

Por lo regular, una base de datos contiene grupos de entidades que son similares. Por ejemplo, una compañía que da empleo a cientos de empleados seguramente querrá almacenar información similar sobre cada uno de ellos. Estas entidades empleado comparten los mismos atributos, pero cada entidad tiene su propio valor (o valores) para cada atributo. Un **tipo de entidades** define un conjunto de entidades que poseen los mismos atributos. Cada tipo de

entidades de la base de datos se describe con un nombre y una lista de atributos. La figura 3.5 muestra dos tipos de entidades, llamados EMPLEADO y COMPAÑÍA, y una lista de atributos para cada uno. También se ilustran algunas entidades individuales de cada tipo, junto con los valores de sus atributos.

Los tipos de entidades se representan en los diagramas ER (como en la Fig. 3.2) por medio de rectángulos que encierran el nombre del tipo de entidades. Los nombres de los atributos se encierran en óvalos y se conectan con su tipo de entidades con líneas rectas. Los atributos compuestos se conectan con sus atributos componentes mediante líneas rectas. Los atributos multivaluados aparecen en óvalos de doble contorno.

Un tipo de entidades describe el **esquema** o la **intensión** para un *conjunto de entidades* que comparten la misma estructura. Las entidades individuales de un tipo de entidades particular se agrupan en una **colección** o **conjunto de entidades**, que se conoce también como **extensión** del tipo de entidades.

Atributos clave de un tipo de entidades. Una restricción importante de las entidades de un tipo es la restricción de **clave** o de **unicidad** de los atributos. Los tipos de entidades casi siempre tienen un atributo cuyo valor es distinto para cada entidad individual. Los atributos de esta naturaleza se denominan **atributos clave**, y sus valores pueden servir para identificar de manera única a cada entidad. El atributo Nombre es una clave del tipo de entidad COMPAÑÍA de la figura 3.5, porque no se permite que dos compañías tengan el mismo nombre. En el caso del tipo de entidades PERSONA, un atributo clave característico es NúmeroDeSeguroSocial. Hay ocasiones en que varios atributos juntos constituyen una clave, o sea que la *combinación* de los valores de los atributos es distinta para cada entidad individual. Un conjunto de atributos que posea esta propiedad se podría agrupar para formar un atributo compuesto, el cual se convertiría en el atributo clave del tipo de entidades. En la notación de los diagramas ER, el nombre de todo atributo clave aparece **subrayado** dentro del óvalo, como se ilustra en la figura 3.2.

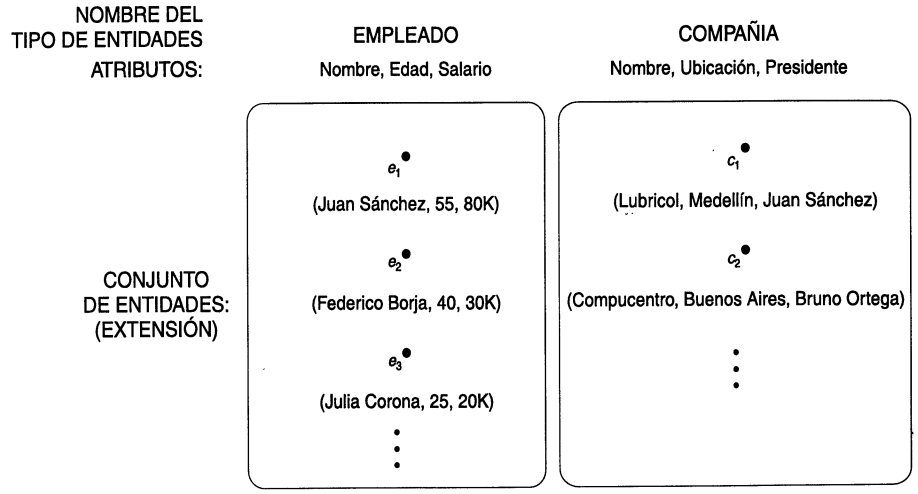


Figura 3.5 Dos tipos de entidades y ejemplos de entidades de cada uno.

Especificar que un atributo es una clave de un tipo de entidades significa que la propiedad de unicidad antes mencionada se debe cumplir para *toda extensión* del tipo de entidades. Por tanto, es una restricción que prohíbe que cualesquiera dos entidades tengan simultáneamente el mismo valor para el atributo clave. No es una propiedad de una extensión específica; más bien, es una restricción para *todas las extensiones* del tipo de entidades. Esta restricción de clave (y otras que veremos más adelante) se deriva de las propiedades del minimundo representado por la base de datos.

Algunos tipos de entidades tienen *más de un* atributo clave. Por ejemplo, tanto el atributo IDVehículo como el atributo Matrícula del tipo de entidades COCHE (Fig. 3.6) son claves por derecho propio. El atributo Matrícula es un ejemplo de clave compuesta formada por dos atributos componentes simples, NúmMatrícula y Estado, ninguno de los cuales es una clave por sí mismo.

Conjuntos de valores (dominios) de los atributos. Cada uno de los atributos simples de un tipo de entidades está asociado a un **conjunto de valores** (o **dominio**), que especifica los valores que es posible asignar a ese atributo para cada entidad individual. En la figura 3.5, si el intervalo de edades permitido para los empleados es de 16 a 70, podemos especificar el conjunto de valores del atributo Edad de EMPLEADO como el conjunto de números enteros entre 16 y 70. De manera similar, podemos especificar el conjunto de valores del atributo Nombre como el conjunto de cadenas de caracteres alfabéticos separadas por caracteres de espacio en blanco; y así sucesivamente. Los conjuntos de valores no se representan en los diagramas ER.

En términos matemáticos, un atributo A de un tipo de entidades E cuyo conjunto de valores es V se puede definir como una **función** de E al conjunto potencia† de V:

$$A : E \rightarrow P(V)$$

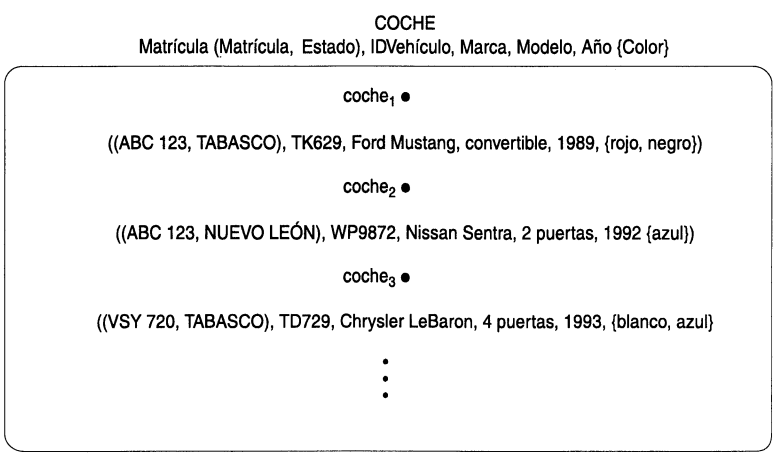


Figura 3.6 El tipo de entidades COCHE. Los atributos multivaluados aparecen entre claves de conjunto { }. Los componentes de un atributo compuesto aparecen entre paréntesis ( ).

†El **conjunto potencia** P(V) de un conjunto V es el conjunto de todos los subconjuntos de V.

Al valor del atributo A para la entidad e lo llamaremos $A(e)$. La definición anterior abarca los atributos monovaluados y multivaluados, además de los nulos. Un valor nulo se representa con el conjunto vacío. En el caso de atributos monovaluados, $A(e)$ sólo puede ser un conjunto unitario† para cada entidad e de E, pero no existe esta restricción para los atributos multivaluados. En el caso de un atributo compuesto A, el conjunto de valores V es el producto cartesiano de $P(V_1)$, $P(V_2)$, ..., $P(V_n)$, donde V_1, V_2, ..., V_n son los conjuntos de valores de los atributos componentes simples que constituyen A:

$$V = P(V_1) \times P(V_2) \times ... \times P(V_n)$$

Cabe señalar que los atributos compuestos y multivaluados pueden estar anidados de cualquier manera. Podemos representar una anidación arbitraria agrupando componentes de un atributo compuesto entre paréntesis ( ) y separando los componentes con comas, y encerrando los atributos multivaluados en claves { }. Por ejemplo, si una persona puede tener más de una residencia y cada residencia puede tener varios teléfonos, se podría especificar un atributo DirecciónTeléfono para un tipo de entidades PERSONA, como en la figura 3.7.

Diseño conceptual inicial de la base de datos COMPAÑÍA. Ahora podemos definir los tipos de entidades para la base de datos COMPAÑÍA descrita en la sección 3.2. Después de definir varios tipos de entidades y sus atributos aquí, *refinaremos* nuestro diseño en la siguiente sección (después de presentar el concepto de vínculo). Según los requerimientos especificados en la sección 3.2, podemos identificar cuatro tipos de entidades, uno para cada uno de los cuatro elementos de la especificación (véase la Fig. 3.8):

1. Un tipo de entidades DEPARTAMENTO con los atributos Nombre, Número, Lugares, Gerente y FechaInicGerente. Lugares es el único atributo multivaluado. Podemos especificar que tanto Nombre como Número sean atributos clave, porque se especificó que ambos debían ser únicos.
2. Un tipo de entidades PROYECTO con los atributos Nombre, Número, Lugar y DeptoControlador. Tanto Nombre como Número son atributos clave.
3. Un tipo de entidades EMPLEADO con los atributos Nombre, NSS (para el número de seguro social), Sexo, Dirección, Salario, FechaNac, Departamento y Supervisor. Tanto Nombre como Dirección pueden ser atributos compuestos, aunque esto no se especificó en los requerimientos. Debemos remitirnos a los usuarios para ver si alguno de ellos va a hacer referencia a los componentes individuales de Nombre —NomPila, Paterno, Materno— o de Dirección.
4. Un tipo de entidades DEPENDIENTE con los atributos Empleado, NombreDependiente, Sexo, FechaNac y Parentesco (con el empleado).

{DirecciónTeléfono({Teléfono(CódÁrea, NúmTeléfono)},
Dirección(Domicilio(Calle, NúmExterior, NúmInterior),
Ciudad, País, CP))}

Figura 3.7 Un atributo compuesto multivaluado, DirecciónTeléfono, con componentes multivaluados y compuestos.

†Un conjunto unitario es un conjunto con un solo elemento (valor).

Hasta aquí, no hemos representado el hecho de que un empleado puede trabajar en varios proyectos, ni el número de horas a la semana que un empleado trabaja en cada proyecto. Esta característica aparece como parte del requerimiento 3 de la sección 3.2, y se puede representar mediante un atributo compuesto multivaluado de EMPLEADO llamado TrabajaEn, con los componentes simples (Proyecto, Horas). Como alternativa, se podría representar mediante un atributo compuesto multivaluado de PROYECTO, llamado Trabajadores, con los componentes simples (Empleado, Horas). En la figura 3.8 elegimos la primera alternativa, donde se muestran todos los tipos de entidades que acabamos de describir. El atributo Nombre de EMPLEADO aparece como atributo compuesto, probablemente como resultado de haber consultado con los usuarios.

En la figura 3.8 hay varios *vínculos implícitos* entre los diversos tipos de entidades. De hecho, siempre que un atributo de un tipo de entidades hace referencia a otro tipo de entidades, hay algún vínculo. Por ejemplo, el atributo Gerente de DEPARTAMENTO se refiere a un empleado que dirige el departamento; el atributo DeptoControlador de PROYECTO se refiere al departamento que controla el proyecto; el atributo Supervisor de EMPLEADO se refiere a otro empleado (el que supervisa a este empleado); el atributo Departamento de EMPLEADO se refiere al departamento para el cual trabaja el empleado, etc. En el modelo ER, estas referencias no se deben representar como atributos, sino como **vínculos**, que se definirán en la siguiente sección. El esquema de la base de datos COMPAÑÍA se refinará en la sección 3.3.5 para representar explícitamente los vínculos. Durante el diseño inicial de los tipos de entidades, los vínculos suelen capturarse en forma de atributos. Al refinarse el diseño, estos atributos se convierten en vínculos entre los tipos de entidades.

3.3.3 *Vínculos, papeles y restricciones estructurales*

Tipos de vínculos y ejemplares de vínculos. Un **tipo de vínculos** R entre n tipos de entidades $E_1, E_2, ..., E_n$ define un conjunto de asociaciones entre entidades de estos tipos. En términos matemáticos, R es un conjunto de **ejemplares de vínculos** r_i, donde cada r_i asocia n entidades $(e_1, e_2, ..., e_n)$ y cada entidad e_j de r_i es miembro del tipo de entidades E_j, $1 \leq j \leq n$. Por tanto, un tipo de vínculos es una relación matemática sobre $E_1, E_2, ..., E_n$, que también puede definirse como un subconjunto del producto cartesiano $E_1 \times E_2 \times \cdots \times E_n$. Se dice que cada uno de los tipos de entidades $E_1, E_2, ..., E_n$ **participa** en el tipo de vínculos R y, de manera similar, que cada una de las entidades individuales $e_1, e_2, ..., e_n$ participa en el ejemplar de vínculo $r_i = (e_1, e_2, ..., e_n)$.

DEPARTAMENTO
Nombre, Número, {Lugares}, Gerente, FechaIngresoGerente

PROYECTO
Nombre, Número, Lugar, DeptoControlador

EMPLEADO
Nombre (NomPila, Paterno, Materno), NSS, Sexo, Dirección, Salario, FechaNac, Departamento, Supervisor, {TrabajaEn (Proyecto, Horas)}

DEPENDIENTE
Empleado, NombreDependiente, Sexo, FechaNac, Parentesco

Figura 3.8 Diseño preliminar de los tipos de entidades para la base de datos descrita en la sección 3.2. Los atributos multivaluados aparecen entre claves { }; los componentes de un atributo compuesto aparecen entre paréntesis ( ).

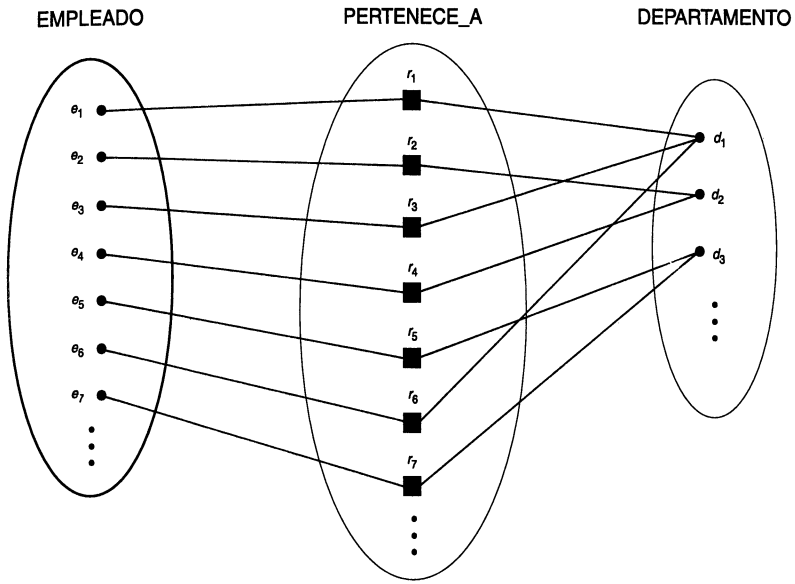


Figura 3.9 Algunos ejemplares del vínculo PERTENECE_A.

En términos informales, cada ejemplar de vínculo r_i de R es una asociación de entidades, donde la asociación incluye una y sólo una entidad de cada tipo de entidades participante. Cada uno de estos ejemplares de vínculos r_i representa el hecho de que las entidades que participan en r_i están relacionadas entre sí de alguna manera en la situación correspondiente del minimundo.

Por ejemplo, consideremos un tipo de vínculos PERTENECE_A entre los dos tipos de entidades EMPLEADO y DEPARTAMENTO, que asocia a cada empleado con el departamento al que pertenece. Cada ejemplar de vínculo de PERTENECE_A asocia una entidad empleado y una entidad departamento. La figura 3.9 ilustra este ejemplo, donde cada ejemplar de vínculo r_i aparece conectado a las entidades departamento y empleado que participan en r_i. En el minimundo que muestra la figura 3.9, los empleados e_1, e_3 y e_6 trabajan en el departamento d_1; e_2 y e_4 trabajan en d_2, y e_5 y e_7 trabajan en d_3.

En los diagramas ER, los tipos de vínculos se representan con rombos conectados mediante líneas rectas con los rectángulos que representan a los tipos de entidades participantes. El nombre del vínculo aparece dentro del rombo (véase la Fig. 3.2).

Grado de un tipo de vínculos. El **grado** de un tipo de vínculos es el número de tipos de entidades que participan en él. Así, el tipo PERTENECE_A es de grado dos. Los tipos de vínculos de grado dos se llaman **binarios**, y los de grado tres se llaman **ternarios**. En la figura 3.10 se muestra un ejemplo de tipo de vínculos ternario, SUMINISTRAR, donde cada ejemplar de vínculo r_i asocia tres entidades —un proveedor v, un componente c y un proyecto p— siempre que v suministre el componente c al proyecto p. Los vínculos pueden tener cualquier grado, pero los más comunes son los binarios. Hablaremos más acerca de los vínculos de mayor grado en la sección 3.6.

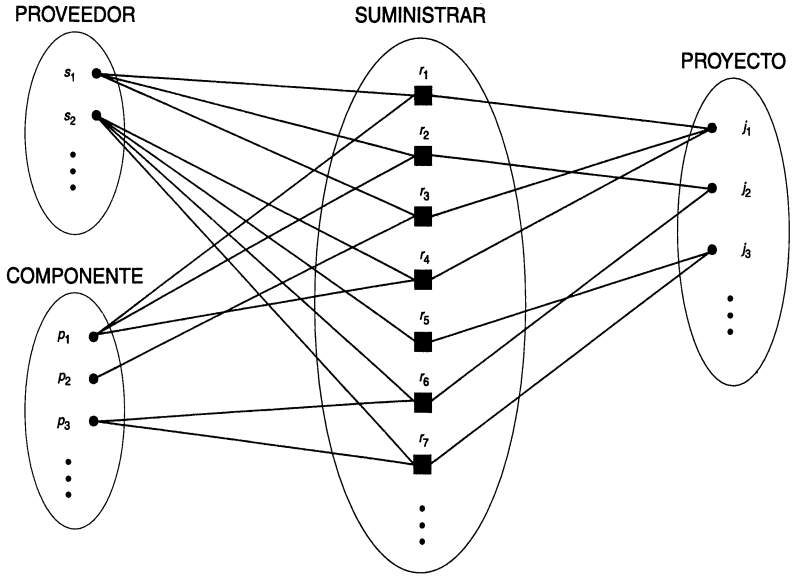


Figura 3.10 El vínculo ternario SUMINISTRAR.

Vínculos como atributos. En ocasiones resulta conveniente considerar un tipo de vínculos en términos de atributos, como vimos al final de la sección anterior. Tomemos como ejemplo el tipo de vínculos PERTENECE_A de la figura 3.9. Podemos pensar en un atributo llamado Departamento del tipo de entidades EMPLEADO, cuyo valor para cada entidad empleado sea la *entidad departamento* a la cual pertenece el empleado. Por tanto, el conjunto de valores de este atributo Departamento es el *conjunto de todas las entidades* DEPARTAMENTO. Esto es lo que hicimos en la figura 3.8, cuando especificamos el diseño inicial del tipo de entidades EMPLEADO para la base de datos COMPAÑÍA. Sin embargo, cuando consideramos un vínculo binario como atributo siempre tenemos dos opciones. En este ejemplo, la alternativa es pensar en un atributo multivaluado Empleados del tipo de entidades DEPARTAMENTO, cuyos valores para cada entidad departamento son *el conjunto de entidades empleado* que trabajan para ese departamento. El conjunto de valores de este atributo Empleados es el conjunto de entidades EMPLEADO. Cualquiera de estos dos atributos —Departamento de EMPLEADO o Empleados de DEPARTAMENTO— puede representar el tipo de vínculos PERTENECE_A. Si ambos están representados, cada uno debe ser el inverso del otro.†

Nombres de papeles y vínculos recursivos. Todo tipo de entidades que participe en un tipo de vínculos desempeña un **papel** específico en el vínculo. El **nombre de papel** indica el papel que una entidad participante del tipo desempeña en cada ejemplar de vínculo. Por ejemplo,

†Este concepto de representar los tipos de vínculos como atributos se utiliza en una clase de modelos de datos llamada **modelos de datos funcionales** (véase la Sec. 21.6.1). En los modelos de datos orientados a objetos (Cap. 22) y semántico (Sec. 21.6.4), los vínculos se pueden representar con *atributos de referencia*, sea en una dirección o en ambas direcciones como inversos. En el modelo de datos relacional (Cap. 6), las *claves externas* son un tipo de atributos de referencia con que se representan vínculos.

en el tipo de vínculos PERTENECE_A, EMPLEADO desempeña el papel de *empleado* o *trabajador* y DEPARTAMENTO tiene el papel de *departamento* o *patrón*.

No son necesarios los nombres de papeles en los tipos de vínculos en los que todos los tipos de entidades participantes son distintos, ya que cada nombre del tipo de entidades se puede usar como nombre del papel. Sin embargo, en algunos casos el *mismo* tipo de entidades participa más de una vez en un tipo de vínculos con *diferentes papeles*. En tales casos el nombre del papel resulta indispensable para distinguir el significado de cada participación. Estos tipos de vínculos se llaman **recursivos**, y la figura 3.11 muestra un ejemplo. El tipo de vínculos SUPERVISIÓN relaciona un empleado con un supervisor, y las entidades empleado y supervisor son ambas miembros del mismo tipo de entidades EMPLEADO. Así, el tipo EMPLEADO *participa dos veces* en SUPERVISIÓN: una vez en el *papel de supervisor* (o jefe), y una vez en el *papel de supervisado* (o subordinado). Cada ejemplar de vínculo r_i en SUPERVISIÓN asocia dos entidades empleado e_j y e_k, una de las cuales desempeña el papel de supervisor y la otra el de supervisado. En la figura 3.11, las líneas marcadascon "1" representan el papel de supervisor y las marcadas con "2" representan el papel de supervisado; por tanto, e_1 supervisa a e_2 y a e_3; e_4 supervisa a e_6 y a e_7, y e_5 supervisa a e_1 y a e_4.

Restricciones sobre los tipos de vínculos. Los tipos de vínculos suelen tener ciertas restricciones que limitan las posibles combinaciones de entidades que pueden participar en los ejemplares de vínculos. Estas restricciones se determinan a partir de la situación del minimundo que los vínculos representan. Por ejemplo, en la figura 3.9, si la compañía tiene una regla de que un empleado sólo puede trabajar para un departamento, nos gustaría describir esta restricción en el esquema. Podemos distinguir dos tipos principales de restricciones de vínculo: razón de cardinalidad y participación.

La **razón de cardinalidad** especifica el número de ejemplares de vínculos en los que puede participar una entidad. El tipo de vínculo binario PERTENECE_A entre DEPARTAMENTO y EMPLEADO tiene razón de cardinalidad 1:*N*, lo que significa que cada departamento puede estar relacionado con muchos empleados, pero un empleado sólo puede estar relacionado con (pertenecer a) un departamento. Las razones de cardinalidad más comunes en el caso de tipos de vínculos binarios son 1:1, 1:*N* y M:*N*.

Un ejemplo de tipo de vínculos 1:1 es DIRIGE (Fig. 3.12), que relaciona una entidad departamento con el empleado que dirige ese departamento. Esto representa las restricciones del minimundo de que un empleado sólo puede dirigir un departamento y de que un departamento sólo tiene un gerente. El tipo de vínculos TRABAJA_EN (Fig. 3.13) tiene razón de cardinalidad M:*N*, si la regla es que un empleado puede trabajar en varios proyectos y que varios empleados pueden trabajar en un proyecto.

La **restricción de participación** especifica si la existencia de una entidad depende de que esté relacionada con otra entidad a través del tipo de vínculos. Hay dos clases de restricciones de participación —total y parcial— que ilustraremos con ejemplos. Si la política de una compañía establece que *todo* empleado debe pertenecer a un departamento, una entidad empleado sólo puede existir si participa en un ejemplar del vínculo PERTENECE_A (Fig. 3.9). Se dice que la participación de EMPLEADO en PERTENECE_A es **total**, porque toda entidad del "conjunto total" de entidades empleado debe estar relacionada con una entidad departamento a través de PERTENECE_A. La participación total recibe a veces el nombre de **dependencia de existencia**. En la figura 3.12 no cabe esperar que todo empleado dirija un departamento, así que la participación de EMPLEADO en el tipo de vínculos DIRIGE es **parcial**, lo que significa que *algunas* o "parte del conjunto de" entidades empleado están relacionadas con una entidad

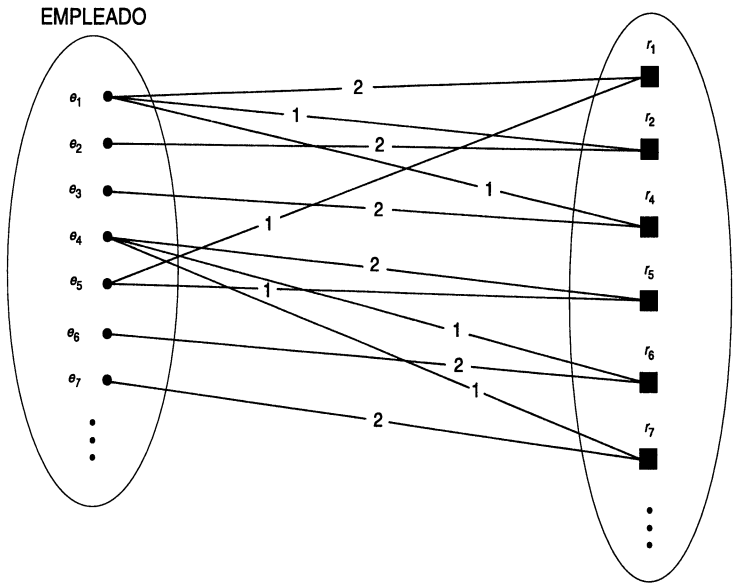


Figura 3.11 El vínculo recursivo SUPERVISIÓN: EMPLEADO desempeña los dos papeles de supervisor (1) y supervisado (2).

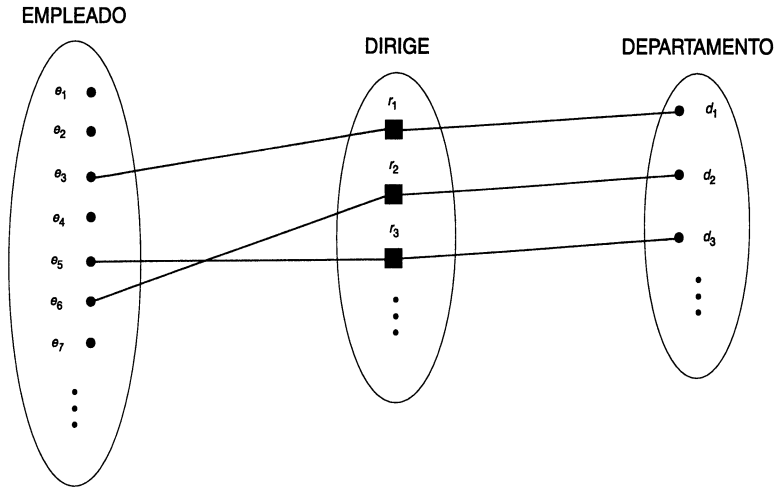


Figura 3.12 El vínculo 1:1 DIRIGE, con participación parcial de EMPLEADO y participación total de DEPARTAMENTO.

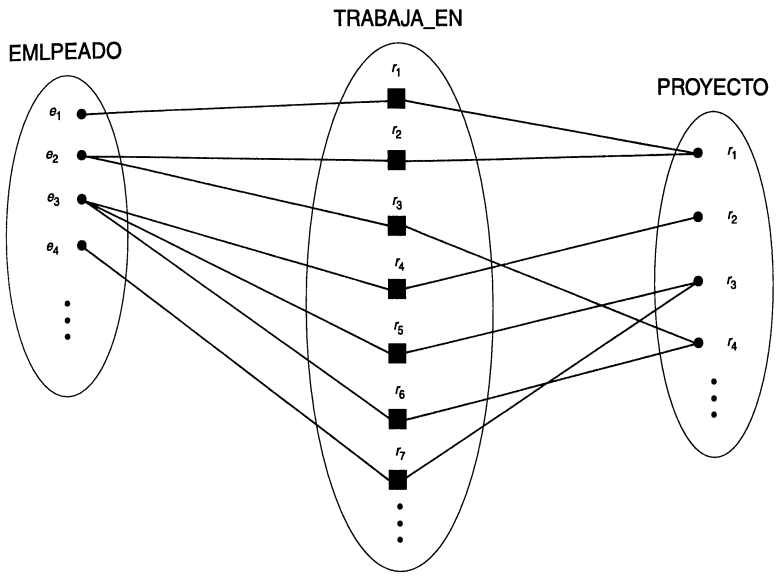


Figura 3.13 El vínculo *M*:*N* TRABAJA_EN.

departamento a través de DIRIGE, pero no necesariamente todas. Nos referiremos a la razón de cardinalidad y a las restricciones de participación, en conjunto, como las **restricciones estructurales** de un tipo de vínculos.

Las razones de cardinalidad de los vínculos binarios se indican en los diagramas ER anotando 1, M y *N*, como en la figura 3.2. La participación total se indica con una línea doble que conecta el tipo de entidades participante con el vínculo, en tanto que la participación parcial se indica con una línea simple.

Atributos de los tipos de vínculos. Los tipos de vínculos también pueden tener atributos, similares a los de los tipos de entidades. Por ejemplo, para registrar el número de horas por semana que un empleado trabaja en un proyecto, podemos incluir un atributo Horas para el tipo de vínculos TRABAJA_EN de la figura 3.13. Otro ejemplo sería incluir la fecha en la que un gerente comenzó a dirigir un departamento, mediante un atributo FechaInic para el tipo de vínculos DIRIGE de la figura 3.12.

Cabe señalar que los atributos de los tipos de vínculos 1:1 o 1:*N* se pueden trasladar a uno de los tipos de entidades participantes. Por ejemplo, el atributo FechaInic del vínculo DIRIGE puede ser atributo tanto de EMPLEADO como de DEPARTAMENTO, aunque conceptualmente pertenece a DIRIGE. La razón es que DIRIGE es un vínculo 1:1, de modo que toda entidad departamento o empleado participará en cuando más un ejemplar de vínculos. Por ende, el valor del atributo FechaInic se puede determinar por separado, ya sea mediante la entidad departamento participante o por la entidad empleado (gerente) participante.

En el caso de un tipo de vínculos 1:*N*, un atributo de éste *sólo* se podrá trasladar al tipo de entidades que está del lado *N* del vínculo. Por ejemplo, en la figura 3.9, si el vínculo PERTENECE_A tiene también un atributo FechaInic que indica cuándo un empleado comenzó

a trabajar para un departamento, dicho atributo se podrá incluir como atributo de EMPLEADO. Esto se debe a que el vínculo es 1:*N*, de modo que cada entidad empleado participa en cuando más un ejemplar de vínculo PERTENECE_A. En el caso de los tipos de vínculos 1:1 y 1:*N*, la decisión de dónde colocar un atributo del vínculo –como atributo del tipo de vínculos o como atributo de un tipo de entidades participante– es algo que debe determinar subjetivamente el diseñador del sistema.

En el caso de los tipos de vínculos M:*N*, algunos atributos pueden estar determinados por la *combinación de las entidades participantes* en un ejemplar de vínculo, y no por alguna de ellas sola. Tales atributos *deberán* especificarse como atributos del vínculo. Un ejemplo es el atributo Horas del vínculo M:*N* TRABAJA_EN (Fig. 3.13); el número de horas que un empleado trabaja en un proyecto lo determina una combinación empleado-proyecto, y no cualquiera de las dos entidades individualmente.

3.3.4 *Tipos de entidades débiles*

Es posible que algunos tipos de entidades no tengan atributos clave propios; éstos se denominan **tipos de entidades débiles**. Las entidades que pertenecen a un tipo de entidades débil se identifican por su relación con entidades específicas de otro tipo de entidades, en combinación con algunos de los valores de sus atributos. Decimos que este otro tipo de entidad es el **propietario identificador**, y llamamos al tipo de vínculos que relaciona un tipo de entidades débil con su propietario el **vínculo identificador** del tipo de entidades débil. Los tipos de entidades débiles siempre tienen una restricción de participación *total* (dependencia de existencia) con respecto a su vínculo identificador, porque una entidad débil no se puede identificar sin una entidad propietaria. Sin embargo, no toda dependencia de existencia resulta en un tipo de entidades débil. Por ejemplo, una entidad LICENCIA_DE_CONDUCTOR no puede existir a menos que esté relacionada con una entidad PERSONA, aunque tenga su propia clave (NÚM_LICENCIA) y por tanto no es una entidad débil.

Consideremos el tipo de entidades DEPENDIENTE, relacionada con EMPLEADO, que sirve para llevar el control de los dependientes de cada empleado a través de un vínculo 1:*N*. Los atributos de DEPENDIENTE son NombreDependiente (el nombre de pila del dependiente), FechaNac, Sexo y Parentesco (con el empleado). Es posible que dos dependientes de empleados distintos tengan los mismos valores de NombreDependiente, FechaNac, Sexo y Parentesco, pero seguirán siendo entidades distintas. Se podrán identificar como entidades distintas sólo después de determinar la entidad empleado con la que está relacionada cada una. Se dice que cada entidad empleado **posee** (o es propietaria de) las entidades dependiente relacionadas con ella.

Por lo regular, los tipos de entidades débiles tienen una **clave parcial**, que es el conjunto de atributos que pueden identificar de manera única las entidades débiles relacionadas con *la misma entidad propietaria*. En nuestro ejemplo, si suponemos que nunca dos dependientes del mismo empleado tendrán el mismo nombre, el atributo NombreDependiente de DEPENDIENTE será la clave parcial.

En los diagramas ER, los tipos de entidades débiles y sus vínculos identificadores se distinguen rodeando los rectángulos y rombos con líneas dobles (véase la Fig. 3.2). El atributo de clave parcial se subraya con una línea punteada o interrumpida.

Hay ocasiones en las que los tipos de entidades débiles se representan en forma de atributos multivaluados compuestos. En el ejemplo anterior, podríamos especificar un atributo multivaluado compuesto Dependientes para EMPLEADO, constituido por los atributos NombreDependiente, FechaNac, Sexo y Parentesco. El diseñador de la base de datos decide cuál representación se utilizará. Una estrategia es elegir la representación de tipo de entidades débil si tiene muchos atributos y participa de manera independiente en otros tipos de vínculos, aparte de su tipo de vínculo identificador. En general, es posible definir cualquier cantidad de niveles de tipos de entidades débiles; un tipo de entidades propietario puede ser él mismo un tipo débil. Además, los tipos de entidades débiles pueden tener más de un tipo de entidades identificador y un tipo de vínculo identificador de grado superior a dos, como veremos en la sección 3.6.

3.3.5 *Refinación del diseño ER para la base de datos COMPAÑÍA*

Ahora podemos refinar el diseño de la base de datos de la figura 3.8 convirtiendo los atributos que representan vínculos en tipos de vínculos. La razón de cardinalidad y la restricción de participación de cada tipo de vínculos se determinan a partir de los requerimientos listados en la sección 3.2. Si no es posible determinar alguna razón de cardinalidad o dependencia a partir de los requerimientos, habrá que consultar con los usuarios para determinar estas propiedades estructurales.

En nuestro ejemplo, especificaremos los siguientes tipos de vínculos:

1. DIRIGE, un tipo de vínculos 1:1 entre EMPLEADO y DEPARTAMENTO. La participación de EMPLEADO es parcial, pero la de DEPARTAMENTO no queda clara a partir de los requerimientos. Consultamos con los usuarios, y éstos nos dicen que un departamento siempre debe tener un gerente, lo que implica participación total. Se asigna el atributo FechaInic a este tipo de vínculos.
2. PERTENECE_A, un tipo de vínculos 1:*N* entre DEPARTAMENTO y EMPLEADO. Ambas participaciones son totales.
3. CONTROLA, un tipo de vínculos 1:*N* entre DEPARTAMENTO y PROYECTO. La participación de PROYECTO es total; luego de consultar con los usuarios, se determina que la participación de DEPARTAMENTO es parcial.
4. SUPERVISIÓN, un tipo de vínculos 1:*N* entre EMPLEADO (en el papel de supervisor) y EMPLEADO (en el papel de supervisado). Los usuarios nos dicen que no todo empleado es un supervisor y no todo empleado tiene un supervisor, de modo que ambas participaciones son parciales.
5. TRABAJA_EN que, después de que los usuarios indican que varios empleados pueden trabajar en un proyecto, resulta ser un tipo de vínculos M:*N* con el atributo Horas. Se determina que ambas participaciones son totales.
6. DEPENDIENTES_DE, un tipo de vínculos 1:*N* entre EMPLEADO y DEPENDIENTE, que también es el vínculo identificador del tipo de entidades débil DEPENDIENTE. La participación de EMPLEADO es parcial, en tanto que la de DEPENDIENTE es total.

Después de especificar los seis tipos de vínculos anteriores, eliminamos de los tipos de entidades de la figura 3.8 todos los atributos que se convirtieron en vínculos durante la refinación. Éstos son Gerente y FechaInicGerente de DEPARTAMENTO; DeptoControlador de PROYECTO; Departamento, Supervisor y TrabajaEn de EMPLEADO, y Empleado de DEPENDIENTE.

Es importante tener el mínimo de redundancia posible cuando diseñemos el esquema conceptual de una base de datos. Si se desea cierta redundancia en el nivel de almacenamiento o en el de vistas de usuario, se puede introducir después, como se explicó en la sección 1.6.1.

3.4 Notación para los diagramas de entidad-vínculo (ER)

En las figuras 3.9 a 3.13 se ilustran los tipos de entidades y de vínculos mostrando sus extensiones: las entidades individuales y los ejemplares de vínculos. En los diagramas ER se hace hincapié en representar los esquemas, no los ejemplares. Esto resulta más útil porque rara vez se modifican los esquemas de base de datos, en tanto que esto se hace a menudo con las extensiones. Además, suele ser más fácil representar el esquema que la extensión de la base de datos, porque es mucho más pequeño.

La figura 3.2 muestra el **esquema ER de base de datos** en forma de diagrama ER. Ahora vamos a estudiar la notación completa para los diagramas ER. Los tipos de entidades como EMPLEADO, DEPARTAMENTO y PROYECTO aparecen en rectángulos. Los tipos de vínculos como PERTENECE_A, DIRIGE, CONTROLA y TRABAJA_EN se muestran en rombos conectados a los tipos de entidades participantes mediante líneas rectas. Los atributos aparecen en óvalos, y cada uno está conectado a su tipo de entidades o de vínculos con una línea recta. Los atributos componentes de un atributo compuesto se conectan al óvalo que representa a este último, como puede verse en el caso del atributo Nombre de EMPLEADO. Los atributos multivaluados aparecen en óvalos dobles, como en el caso del atributo Lugares de DEPARTAMENTO. Los nombres de los atributos clave están subrayados. Los nombres de los atributos derivados aparecen en óvalos punteados, como en el caso del atributo NúmDeEmpleados de DEPARTAMENTO.

Los tipos de entidades débiles se distinguen porque sus rectángulos tienen doble borde y porque los vínculos que los identifican están en rombos dobles, como en el caso del tipo de entidades DEPENDIENTE y el tipo de vínculos identificador DEPENDIENTES_DE. La clave parcial del tipo de entidades débil está subrayado con una *línea punteada*.

En la figura 3.2 la razón de cardinalidad de todos los tipos de vínculos *binarios* se especifica anotando 1, M o *N* en cada vértice participante. La razón de cardinalidad de DEPARTAMENTO:EMPLEADO en DIRIGE es 1:1, pero la de DEPARTAMENTO:EMPLEADO en PERTENECE_A es 1:*N*, y en el caso de TRABAJA_EN es M:*N*. La restricción de participación se especifica con una línea simple en el caso de la participación parcial y con una línea doble en el de la participación total (dependencia de existencia).

En la figura 3.2 mostramos los nombres de papeles del tipo de vínculos SUPERVISIÓN porque el tipo de entidades EMPLEADO desempeña ambos papeles en ese vínculo. Obsérvese que la cardinalidad es 1:*N* de supervisor a supervisado porque, por un lado, cada empleado en el papel de supervisado tiene cuando más un supervisor directo, en tanto que un empleado en el papel de supervisor puede supervisar a cero o más empleados.

Una notación alternativa de ER con la que también se pueden especificar las restricciones estructurales consiste en asociar un par de números enteros (*mín*, *máx*) a cada *participación* de un tipo de entidades *E* en un tipo de vínculos *R*, donde $0 \leq mín \leq máx$ y $máx \geq 1$. Los números significan que, para cada entidad *e* de *E*, *e* debe participar en por lo menos *mín* y cuando más en *máx* ejemplares de vínculos de *R en todo momento*. En este método, *mín* = 0 implica participación parcial, en tanto que *mín* > 0 implica participación total. En la figura 3.14, que muestra el esquema COMPAÑÍA, se emplea esta notación. Este método es más

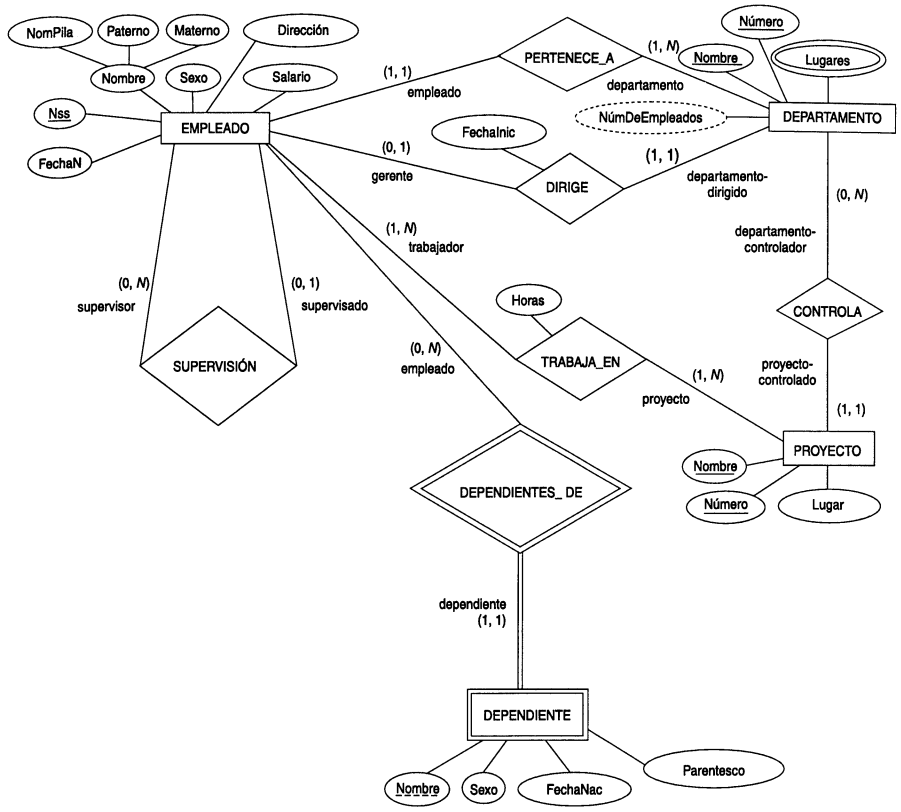


Figura 3.14 Diagrama ER del esquema COMPAÑÍA, con todos los nombres de papeles y las restricciones estructurales de los vínculos con la notación alternativa (*mín*, *máx*).

preciso y fácil de usar para especificar restricciones estructurales de tipos de vínculos *de cualquier grado*. Hay otras notaciones diagramáticas para representar los diagramas ER, y en el apéndice A se ilustrarán algunas de las más populares.

La figura 3.14 muestra también todos los nombres de papeles del esquema de la base de datos COMPAÑÍA. La figura 3.15 resume las convenciones de los diagramas ER.

3.5 Nombres apropiados para los elementos de esquema*

No siempre es trivial la elección de nombres para los tipos de entidades, los atributos, los tipos de vínculos y (sobre todo) los papeles. Debemos elegir nombres que comuniquen, hasta donde sea posible, los significados conferidos a los distintos elementos de esquema. Optamos por usar *nombres en singular* para los tipos de entidades, y no en plural, porque el nombre del tipo de entidades se aplica a cada una de las entidades individuales que pertenecen

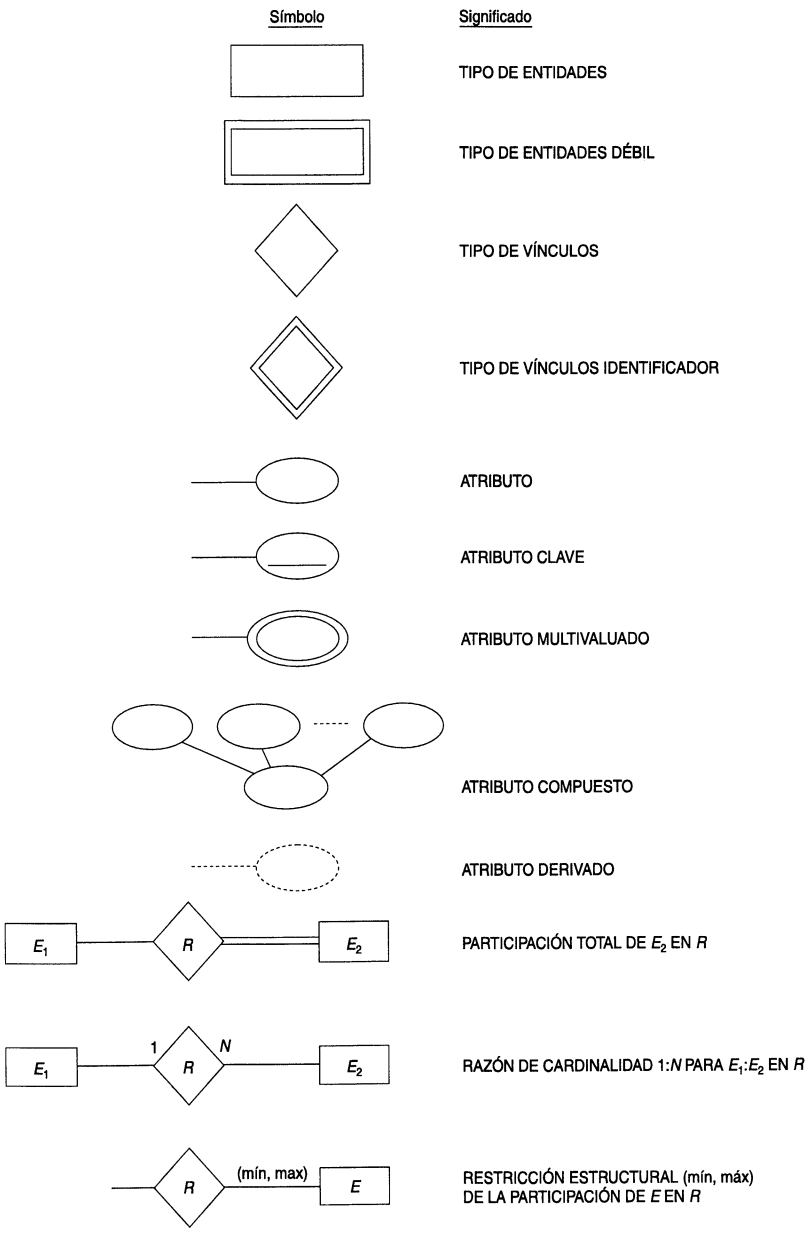


Figura 3.15 Resumen de la notación de diagramas ER.

a ese tipo. En nuestros diagramas ER aplicaremos la convención de que los nombres de los tipos de entidades y de vínculos van en mayúsculas, los nombres de atributos comienzan con mayúscula, y los nombres de papeles van en minúsculas. Ya hemos aplicado esta convención en las figuras 3.2 y 3.14.

Como práctica general, dada una descripción narrativa de los requerimientos de la base de datos, los *sustantivos* que aparezcan en la narración tenderán a originar nombres de tipos de entidades, y los *verbos* tenderán a indicar nombres de tipos de vínculos. Los nombres de los atributos generalmente surgen de los sustantivos adicionales que describen a los sustantivos correspondientes a los tipos de entidades. Otra consideración en lo tocante a los nombres es que los de los vínculos deben elegirse de modo que el diagrama ER del esquema se pueda leer de izquierda a derecha y de arriba hacia abajo. En la figura 3.2 seguimos en términos generales esta pauta, con la excepción del tipo de vínculos DEPENDIENTES_DE, que se lee de abajo hacia arriba. Esto se debe a que decimos que las entidades DEPENDIENTE (el tipo de entidades de abajo) son DEPENDIENTES_DE (nombre de vínculo) un EMPLEADO (tipo de entidades de arriba). Si quisiéramos modificar esto para que se leyera de arriba hacia abajo, podríamos cambiar el nombre del tipo de vínculos a TIENE_DEPENDIENTES, que entonces se leería: una entidad EMPLEADO (tipo de entidades de arriba) TIENE_DEPENDIENTES (nombre de vínculos) del tipo DEPENDIENTE (tipo de entidades de abajo).

3.6 Tipos de vínculos con grado mayor que dos*

En la sección 3.3.3 definimos el **grado** de un tipo de vínculos como el número de tipos de entidades participantes, y dijimos que un tipo de vínculos de grado dos era **binario** y uno de grado tres era **ternario**. La notación de diagrama ER para un tipo de vínculos ternario se ilustra en la figura 3.16(a), que muestra el esquema del tipo de vínculos SUMINISTRAR ilustrado a nivel de ejemplar en la figura 3.10. En general, un tipo de vínculos R de grado n tendrá n aristas conectadas en un diagrama ER, y cada una conectará a R con un tipo de entidades participante.

La figura 3.16(b) muestra un diagrama ER para los tres tipos de vínculos binarios PUEDE_SUMINISTRAR, UTILIZA y SUMINISTRA. En general, un tipo de vínculos ternario representa más información que tres tipos de vínculos binarios. Consideremos los tres tipos de vínculos PUEDE_SUMINISTRAR, UTILIZA y SUMINISTRA. Suponga que PUEDE_SUMINISTRAR, entre PROVEEDOR y COMPONENTE, incluye un ejemplar (v, c) siempre que el proveedor v *puede suministrar* el componente c (a cualquier proyecto); UTILIZA, entre PROYECTO y COMPONENTE, incluye un ejemplar (p, c) siempre que el proyecto p *utiliza* el componente c, y SUMINISTRA, entre PROVEEDOR y PROYECTO, incluye un ejemplar (v, p) siempre que un proveedor v *suministra algún componente* al proyecto p. La existencia de tres ejemplares de vínculos (v, c), (p, c) y (v, p) en PUEDE_SUMINISTRAR, UTILIZA y SUMINISTRA, respectivamente, ¡no necesariamente implica que exista un ejemplar (v, p, c) en el vínculo ternario SUMINISTRAR! A menudo resulta difícil decidir si un cierto vínculo se debe representar como un tipo de vínculos de grado n o si se debe descomponer en varios tipos de vínculos de menor grado. El diseñador debe basar su decisión en la semántica o significado de la situación específica que se va a representar. La solución más común es incluir el vínculo ternario *junto con* uno o más de los vínculos binarios, según se necesite.

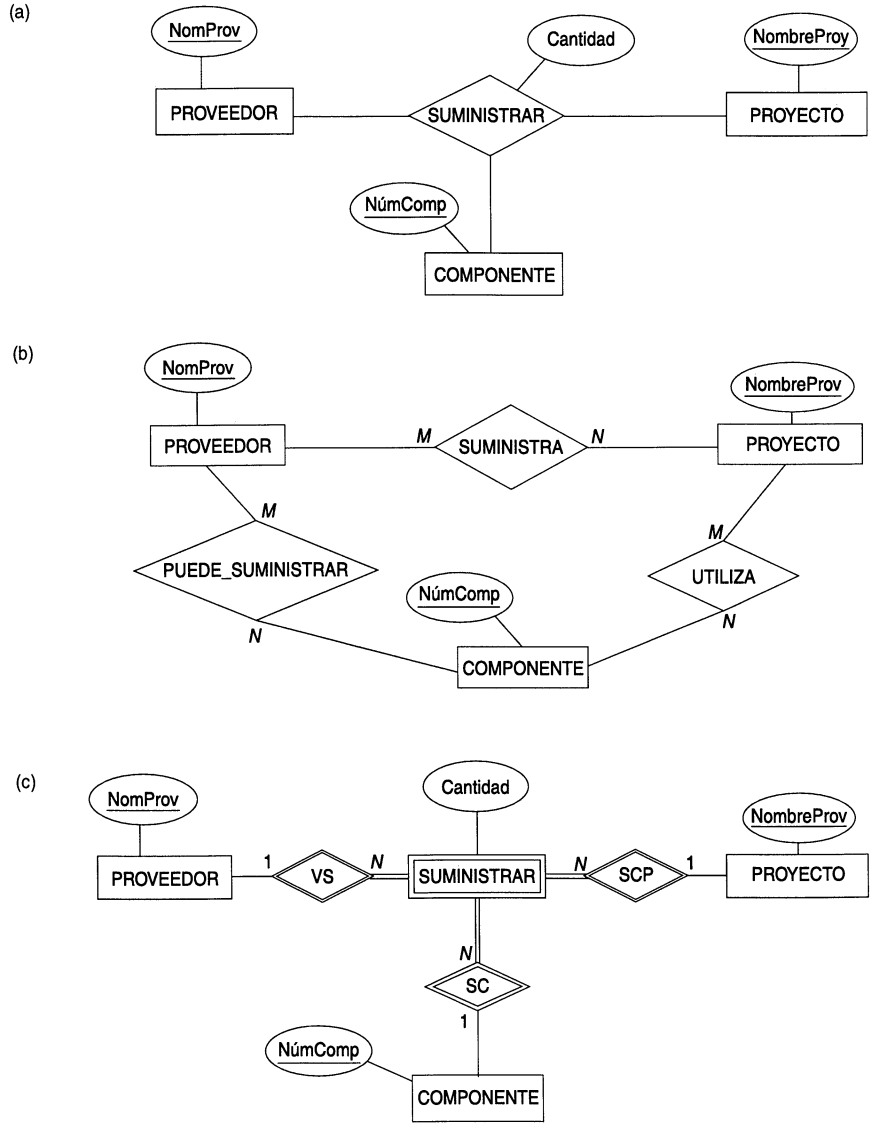


Figura 3.16 Tipos de vínculos ternarios. (a) El tipo de vínculos ternario SUMINISTRAR. (b) Tres tipos de vínculos binarios que no son equivalentes al tipo de vínculos ternario SUMINISTRAR. (c) SUMINISTRAR representado como tipo de entidades débil.

Algunas de las herramientas que se emplean en el diseño de bases de datos tienen su fundamento en variaciones del modelo ER que sólo permiten vínculos binarios. En este caso, los vínculos ternarios como SUMINISTRAR se deben representar como tipos de entidades débiles, sin clave parcial y con tres vínculos identificadores. Los tres tipos de entidades participantes PROVEEDOR, COMPONENTE y PROYECTO, en conjunto, son los tipos de entidades propietarios (véase la Fig. 3.16(c)); por tanto, una entidad del tipo de entidades débil SUMINISTRAR de la figura 3.16(c) se identifica mediante la combinación de sus tres entidades propietarias provenientes de PROVEEDOR, COMPONENTE y PROYECTO.

Podemos ver otro ejemplo en la figura 3.17. El tipo de vínculos ternario OFRECE representa información sobre los profesores que ofrecen cursos durante ciertos semestres; por tanto, incluye un ejemplar de vínculos (p, s, c) siempre que el profesor p ofrece el curso c durante el semestre s. Los tres tipos de vínculos binarios que se muestran en la figura 3.17 tienen los significados siguientes: PUEDE_IMPARTIR relaciona un curso con los profesores que *pueden impartirlo*; IMPARTIÓ_DURANTE relaciona un semestre con los profesores que impartieron *algún curso* durante ese semestre, y SE_OFRECE_DURANTE relaciona un semestre con los cursos ofrecidos durante ese semestre *por cualquier profesor*. En general, estos vínculos ternarios y binarios representan diferente información, pero se deberán mantener ciertas restricciones entre ellos. Por ejemplo, no deberá existir un ejemplar de vínculo (p, s, c) en OFRECE *si no existe* un ejemplar (p, s) en IMPARTIÓ_DURANTE, un ejemplar (s, c) en SE_OFRECE_ DURANTE y un ejemplar (p, c) en PUEDE_IMPARTIR. Sin embargo, lo opuesto no siempre se cumple; podemos tener ejemplares (p, s), (s, c) y (p, c) en los tres tipos de vínculos binarios sin que haya un caso (p, s, c) correspondiente en OFRECE. Bajo ciertas *restricciones adicionales*, esto último puede ser válido; por ejemplo, si el vínculo PUEDE_OFRECER es 1:1 (un profesor sólo puede impartir un curso, y un curso puede ser impartido por sólo un profesor). El diseñador del esquema debe analizar cada situación específica para decidir cuáles de los tipos de vínculos binarios y ternarios se necesitan.

Adviértase que es posible tener un tipo de entidades débil con un tipo de vínculos identificador ternario (o n-ario). En este caso, el tipo de entidades débil puede tener *varios* tipos de entidades propietarios. En la figura 3.18 se muestra un ejemplo.

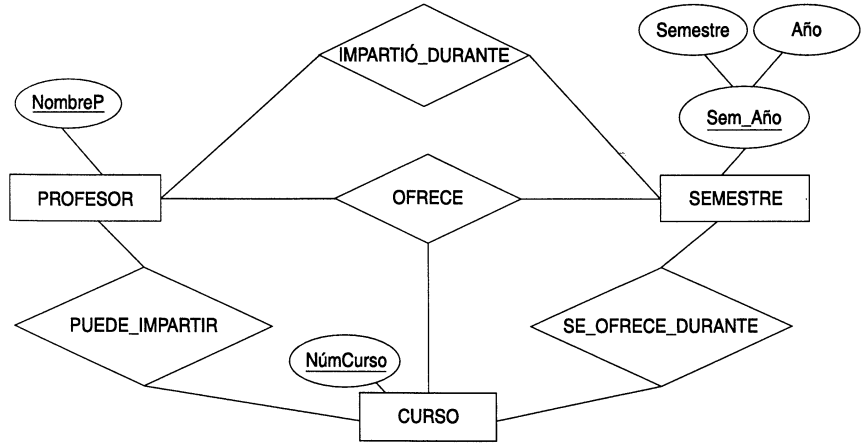


Figura 3.17 Otro ejemplo de tipos de vínculos ternarios vs. binarios.

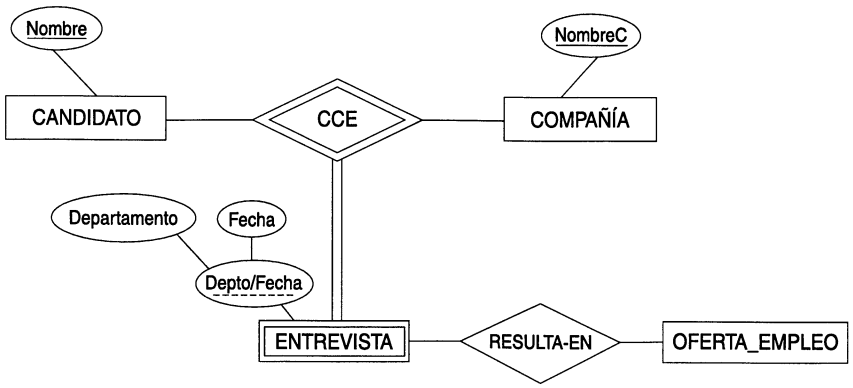


Figura 3.18 Tipo de entidades débil ENTREVISTA, con un tipo de vínculos identificador ternario.

3.7 Resumen

En este capítulo presentamos los conceptos de modelado de un modelo conceptual de datos de alto nivel: el modelo de entidad-vínculo (ER). Comenzamos por analizar el papel que un modelo de datos de alto nivel desempeña en el proceso de diseño de bases de datos, y luego presentamos un ejemplo de conjunto de requerimientos para una base de datos. A continuación definimos los conceptos básicos del modelo ER: las entidades y sus atributos. Tratamos los valores nulos y luego examinamos los diferentes tipos de atributos, que se pueden anidar arbitrariamente:

- Simples o atómicos.
- Compuestos.
- Multivaluados.

También analizamos brevemente los atributos derivados. Después tratamos los conceptos del modelo ER en el nivel de esquema o de "intensión":

- Tipos de entidades y sus conjuntos de entidades correspondientes.
- Atributos y sus conjuntos de valores.
- Atributos clave.
- Tipos de vínculos y sus conjuntos de ejemplares correspondientes.
- Papeles de participación de los tipos de entidades en los tipos de vínculos.

Presentamos dos métodos para especificar las restricciones estructurales de los tipos de vínculos. En el primer método se distinguieron dos tipos de restricciones estructurales:

- Razones de cardinalidad (1:1, 1:*N*, *M*:*N* para vínculos binarios)
- Restricciones de participación (total, parcial).

Señalamos que una alternativa era el método más general de especificar las restricciones estructurales mediante números mínimo y máximo (*mín*, *máx*) de la participación de cada tipo de entidades en un tipo de vínculos. Esto se aplica a los tipos de vínculos de cualquier grado. En seguida tratamos los tipos de entidades débiles y los conceptos relacionados de tipos de entidades propietarios, tipos de vínculos identificadores, y atributos de clave parcial. Los esquemas ER se pueden representar en forma de diagramas ER. Explicamos cómo diseñar un esquema ER para la base de datos COMPAÑÍA definiendo en primer término los tipos de entidades y sus atributos y refinando luego el diseño para incluir los tipos de vínculos. Mostramos el diagrama ER del esquema de la base de datos COMPAÑÍA. Por último, analizamos los tipos de vínculos ternarios y de grado mayor, y las circunstancias en las que se distinguen de un conjunto de tipos de vínculos binarios.

Los conceptos de modelado ER que hemos visto hasta ahora —tipos de entidades, tipos de vínculos, atributos, claves y restricciones estructurales— pueden modelar una gran variedad de aplicaciones de base de datos; sin embargo, algunas aplicaciones —sobre todo algunas de las más nuevas, como las bases de datos para diseño en ingeniería o para aplicaciones de inteligencia artificial— requieren conceptos adicionales si lo que se quiere es un modelado más exacto. Trataremos estos conceptos avanzados en el capítulo 21.

Preguntas de repaso

3.1. Analice el papel que tiene un modelo de datos de alto nivel en el proceso de diseño de bases de datos.

3.2. Mencione los diversos casos en los que sería apropiado usar un valor nulo.

3.3. Defina los siguientes términos: *entidad, atributo, valor de atributo, ejemplar de vínculo, atributo compuesto, atributo multivaluado, atributo derivado, atributo clave, conjunto de valores (dominio)*.

3.4. ¿Qué es un tipo de entidades? ¿Qué es un conjunto de entidades? Explique las diferencias entre una entidad, un tipo de entidades y un conjunto de entidades.

3.5. Explique la diferencia entre un atributo y un conjunto de valores.

3.6. ¿Qué es un tipo de vínculos? Explique la diferencia entre un ejemplar de vínculo y un tipo de vínculos.

3.7. ¿Qué es un papel de participación? ¿Cuándo es *necesario* especificar nombres de papeles en la descripción de los tipos de vínculos?

3.8. Describa las dos alternativas para especificar las restricciones estructurales para los tipos de vínculos. ¿Qué ventajas y desventajas tiene cada una de ellas?

3.9. ¿En qué condiciones se puede trasladar un atributo de un tipo de vínculos binario para convertirse en un atributo de uno de los tipos de entidades participantes?

3.10. Cuando contemplamos los vínculos como atributos, ¿cuáles son los conjuntos de valores de estos atributos? ¿Qué clase de modelos de datos se basa en este concepto?

3.11. ¿Qué quiere decir tipo de vínculos recursivo? Cite algunos ejemplos de tipos de vínculos recursivos.

3.12. ¿En qué casos resulta útil el concepto de entidad débil en el modelado de datos? Defina los términos *tipo de entidades propietario, tipo de entidades débil, tipo de vínculos identificador* y *clave parcial.*

3.13. Un vínculo identificador de un tipo de entidades débil, ¿puede ser de grado mayor que 2? Cite ejemplos.

3.14. Analice las convenciones que se siguen para representar un esquema ER en forma de diagrama ER.

3.15. Analice las condiciones en las que un tipo de vínculos ternario se puede representar con varios tipos de vínculos binarios.

Ejercicios

3.16. Considere el siguiente conjunto de requerimientos para una base de datos universitaria que sirve para manejar las boletas de notas de los estudiantes. Esto es similar, pero no idéntico, a la base de datos de la figura 1.2:

a. Para cada estudiante, la universidad mantiene información que comprende su nombre, su número de estudiante, su número de seguro social, su dirección y número telefónico actuales, su dirección y número telefónico permanentes, su fecha de nacimiento, su sexo, su grado (primero, segundo, ..., posgrado), su departamento de carrera, su departamento de especialidad (si lo hay) y su nivel de estudios (bachillerato en ciencias, bachillerato en humanidades, ..., doctorado). Algunas aplicaciones de los usuarios tendrán que hacer referencia a la ciudad, estado y código postal de la dirección permanente del estudiante, y al apellido de este último. Tanto el número de seguro social como el número de estudiante tienen valores únicos para cada estudiante.

b. Cada departamento se describe mediante un nombre, un código de departamento, un número de oficina, un teléfono de oficina y una facultad. Tanto el nombre como el código tienen valores únicos para cada departamento.

c. Cada curso tiene un nombre de curso, una descripción, un número de curso, un número de horas por semestre, un nivel y un departamento que lo ofrece. El valor del número de curso es único para cada curso.

d. Cada sección tiene un profesor, un semestre, un año, un curso y un número de sección. El número de sección distingue las diferentes secciones de un mismo curso que se imparten durante el mismo semestre/año; sus valores son 1, 2, 3, ..., hasta el número de secciones impartidas durante cada semestre.

e. Un informe de notas tiene un estudiante, una sección, una nota de letra y una nota numérica (0, 1, 2, 3 o 4).

Diseñe un esquema ER para esta aplicación, y dibuje un diagrama ER para ese esquema. Especifique los atributos clave de cada tipo de entidades y las restricciones estructurales de cada tipo de vínculos. Tome nota de cualesquier requerimientos que no se hayan especificado, y haga suposiciones apropiadas para que la especificación sea completa.

3.17. Los atributos compuestos y multivaluados pueden anidarse con cualquier cantidad de niveles. Suponga que desea diseñar un atributo para un tipo de entidades ESTUDIANTE que registre la educación universitaria previa. El atributo en cuestión tendrá una entrada por cada universidad a la que se haya asistido antes, y cada una de esas entradas consistirá en un nombre de universidad, fechas inicial y final, entradas de grado (grados obtenidos en esa universidad, si se obtuvo alguno) y entradas de boleta de notas (cursos concluidos en esa universidad, si se completó alguno). Cada entrada de grado se compondrá del nombre del grado y del mes y año en que éste fue otorgado; cada entrada de boleta de notas consistirá en un nombre de curso, un semestre, un año y una nota. Diseñe un atributo para guardar esta información. Utilice las convenciones de la figura 3.8.

3.18. Muestre otro diseño para el atributo descrito en el ejercicio 3.17 que sólo utilice tipos de entidades (incluidos tipos de entidades débiles, si es preciso) y tipos de vínculos.

3.19. Considere el diagrama ER de la figura 3.19, que muestra un esquema simplificado para el sistema de reservaciones de una línea aérea. Extraiga del diagrama ER los requerimientos y restricciones que produjeron dicho esquema. Trate de especificarlos con la mayor precisión posible.

3.20. En los capítulos 1 y 2 analizamos el entorno de bases de datos y sus usuarios. Podemos considerar muchos tipos de entidades para describir un entorno así: como SGBD, base de datos almacenada, DBA y catálogo/diccionario de datos. Intente especificar todos los tipos de entidades que puedan describir por completo un sistema de base de datos y su entorno, luego especifique los tipos de vínculos que hay entre ellos y dibuje un diagrama ER que describa semejante entorno general de bases de datos.

3.21. Una compañía transportista, llamada CAMIONES, se encarga de recoger embarques en las bodegas de una cadena de ventas al detalle llamada HNOS. VELÁZQUEZ y de entregar tales embarques a los establecimientos de ventas al detalle de HNOS. VELÁZQUEZ. De momento hay 6 bodegas y 45 establecimientos de ventas de HNOS. VELÁZQUEZ. Un camión puede transportar varios embarques en un mismo viaje, identificado por NúmViaje, y entregarlos a múltiples establecimientos. Cada embarque se identifica con NúmEmbarque e incluye datos sobre el peso, el volumen, el destino, etc., del embarque. Los camiones tienen diferentes capacidades tanto para los volúmenes que pueden contener como para los pesos que pueden transportar. La compañía CAMIONES cuenta actualmente con 150 camiones, cada uno de los cuales efectúa tres o cuatro viajes a la semana. Se está diseñando una base de datos —que utilizarán tanto CAMIONES como HNOS. VELÁZQUEZ— para llevar el control del uso de camiones y de las entregas, y que ayude a programar los camiones de manera que realicen entregas oportunas a los establecimientos. Diseñe un diagrama de esquema ER para esta aplicación. Haga todas las suposiciones que necesite, expresándolas con toda claridad.

3.22. Se está construyendo una base de datos para llevar la organización de los equipos y los juegos de una liga deportiva. Cada equipo tiene varios jugadores, aunque no todos participan en un juego dado. Se desea llevar el control de los jugadores que participan en cada juego por parte de cada equipo, de las posiciones que ocuparon en el juego y del resultado del mismo. Intente diseñar un diagrama de esquema ER para esta aplicación, expresando todas las suposiciones que haga. Escoja su deporte favorito (fútbol, béisbol, baloncesto, ...).

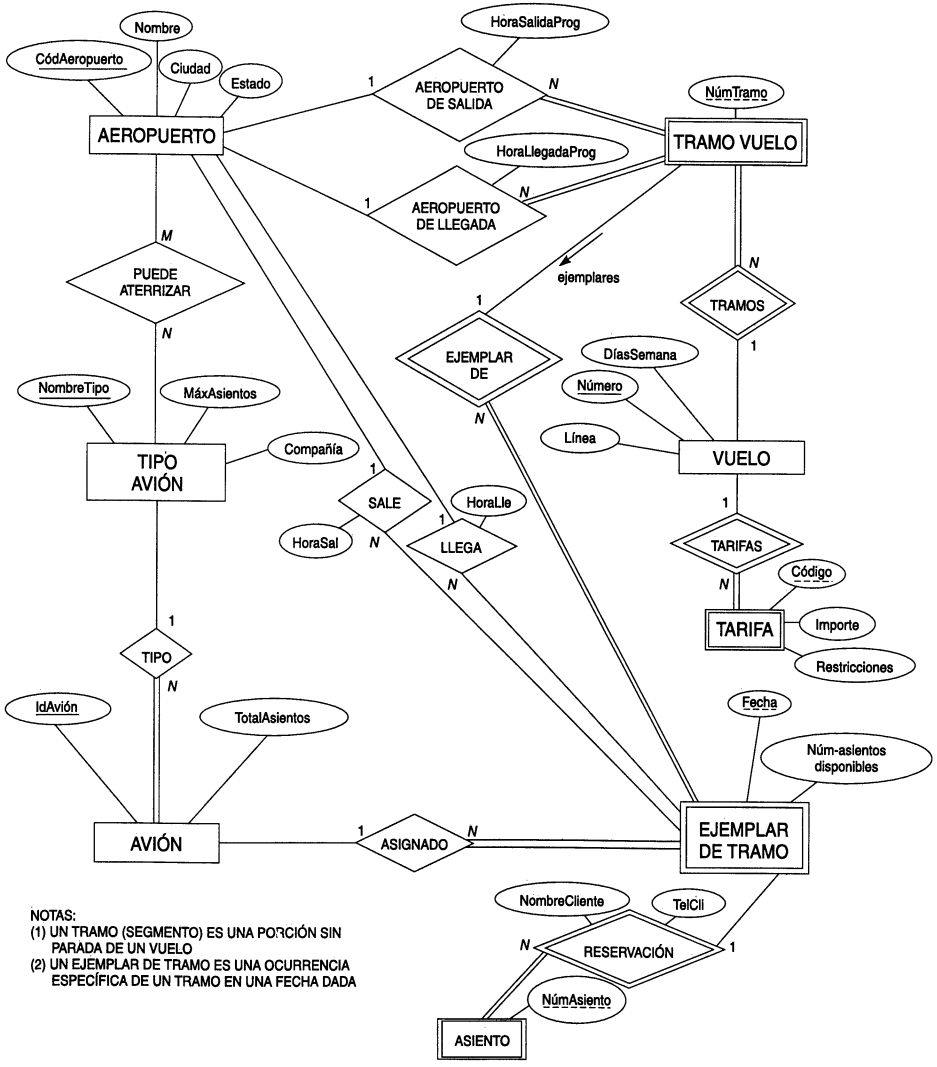


Figura 3.19 Esquema para una línea aérea.

3.23. Considere el diagrama ER de la figura 3.20 para una parte de una base de datos llamada BANCO. Cada banco puede tener múltiples sucursales, y cada sucursal puede tener varias cuentas y préstamos.

a. Haga una lista con todos los tipos de entidades (no débiles) del diagrama.

b. ¿Hay algún tipo de entidades débil? Si es así, indique su nombre, su clave parcial y su vínculo identificador.

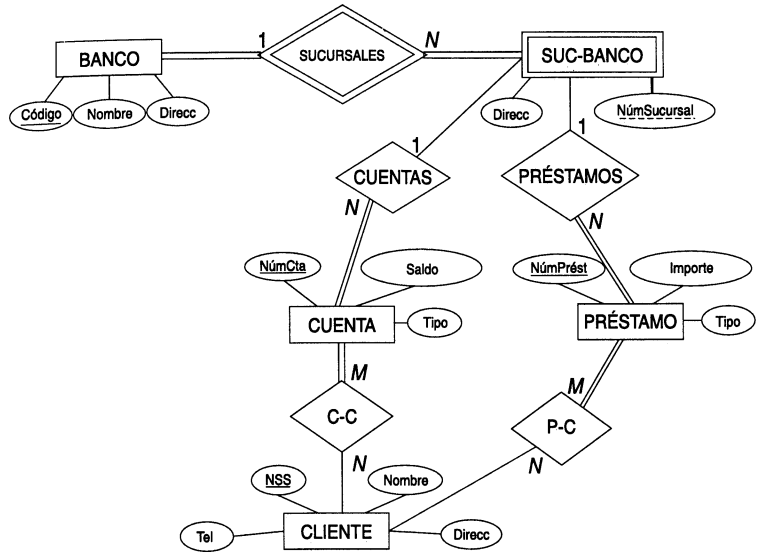


Figura 3.20 Diagrama ER de una base de datos BANCO.

c. ¿Qué restricciones especifican en este diagrama la clave parcial y el vínculo identificador del tipo de entidades débil?

d. Haga una lista con todos los tipos de vínculos y especifique la restricción (*mín*, *máx*) de cada participación de un tipo de entidades en un tipo de vínculos. Justifique sus elecciones.

e. Haga una lista concisa con los requerimientos de usuario que llevaron a este diseño de esquema.

f. Suponga que todo cliente debe tener por lo menos una cuenta pero no puede tener más de dos préstamos simultáneos, y que una sucursal bancaria no puede tener más de 1000 préstamos. ¿Cómo se indica esto en las restricciones (*mín*, *máx*)?

Bibliografía selecta

Chen (1976) propuso el modelo de entidad-vínculo, y en Schmidt y Swenson (1975), Wiederhold y Elmasri (1979) y Senko (1975) aparecen trabajos relacionados. Desde entonces, se ha sugerido un gran número de modificaciones al modelo ER, y hemos incorporado algunas de ellas en nuestra exposición. Las restricciones estructurales de los vínculos se analizan en Abrial (1974), Elmasri y Wiederhold (1980) y Lenzerini y Santucci (1983). Los atributos multivaluados y compuestos se incorporaron en el modelo ER en Elmasri *et al.* (1985), donde también se estudian las operaciones de actualización de ER y la especificación de transacciones. A partir de 1979, se ha celebrado con regularidad una conferencia para diseminar los resultados de investigaciones relacionadas con el modelo ER. Dicha conferencia ha tenido lugar en Los Angeles (ER 1979, ER 1983), Washington, D.C. (ER 1981), Chicago (ER 1985), Dijon, Francia (ER 1986), Nueva York (ER 1987), Roma, (ER 1988), Toronto, Canadá (ER 1989), Lausana, Suiza (ER 1990), San Mateo, California (ER 1991), y Karlsruhe, Alemania (ER 1992).

CAPÍTULO 4

Almacenamiento de registros y organizaciones primarias de archivos

En los capítulos 4 y 5 nos ocuparemos de las técnicas de almacenamiento físico de los datos en el sistema de computador. Las bases de datos suelen estar organizadas en archivos de registros, los cuales se almacenan en discos de computador. Lo primero que haremos, en la sección 4.1, será presentar los conceptos de las jerarquías de almacenamiento en computador. En la sección 4.2 describiremos los dispositivos de almacenamiento en disco y sus características, y también estudiaremos brevemente los dispositivos de almacenamiento en cinta. La sección 4.3 cubre la técnica de doble almacenamiento intermedio, que sirve para agilizar la obtención de múltiples bloques del disco. En la sección 4.4 analizaremos diversas formas de dar formato a los registros y almacenarlos en archivos en disco. En la sección 4.5 presentaremos los diversos tipos de operaciones que suelen aplicarse a los registros de los archivos. Luego hablaremos de tres métodos principales para organizar los registros de un archivo en disco: registros no ordenados (Sec. 4.6), registros ordenados (Sec. 4.7) y registros dispersos (Sec. 4.8).

En la sección 4.9 haremos un breve estudio de los archivos de registros mixtos y otros métodos primarios para organizar los registros, como los árboles B. En el capítulo 5 trataremos las técnicas para crear estructuras de acceso (los índices) que agilizan la búsqueda y obtención de registros. Estas técnicas implican el almacenamiento de datos auxiliares, además de los registros mismos.

Los lectores que ya hayan estudiado las organizaciones de archivos pueden estudiar superficialmente los capítulos 4 y 5, o incluso pasarlos completamente por alto. También pueden posponerse para su lectura posterior. El material que contienen es necesario para entender algunos de los capítulos del libro que vienen después, en particular los capítulos 14 y 16 al 19.

4.1 Introducción

La colección de datos que constituye una base de datos computarizada debe estar almacenada físicamente en algún **medio de almacenamiento** en computador. Así, el software del SGBD podrá leer, actualizar y procesar estos datos cuando sea necesario. Los medios de almacenamiento en computador forman una *jerarquía de almacenamiento* que incluye dos categorías principales:

- Almacenamiento primario. En esta categoría caben medios de almacenamiento sobre los cuales la unidad central de proceso (UCP) del computador puede operar directamente, como la memoria principal y las memorias caché, que son más pequeñas pero más rápidas. Por lo regular, el almacenamiento primario ofrece acceso rápido a los datos, aunque su capacidad de almacenamiento es limitada.
- Almacenamiento secundario. Entre los dispositivos de almacenamiento secundario están los discos magnéticos y ópticos, las cintas y los tambores magnéticos, que casi siempre tienen mayor capacidad, cuestan menos y ofrecen acceso más lento a los datos que los dispositivos de almacenamiento primario. La UCP no puede procesar directamente los datos en almacenamiento secundario; para ello deben copiarse en almacenamiento primario.

Las bases de datos suelen almacenar grandes cantidades de datos que deben persistir por periodos largos. Durante dichos periodos, los datos se leen y procesan una y otra vez. Esto contrasta con la noción de estructuras de datos que persisten sólo por un tiempo limitado durante la ejecución de un programa, cosa que es común en los lenguajes de programación. La mayoría de las bases de datos se almacenan permanentemente en almacenamiento secundario de **disco magnético**, por las siguientes razones:

- En general, las bases de datos son demasiado grandes como para que quepan completas en la memoria principal.
- Las circunstancias que provocan la pérdida permanente de los datos almacenados se presentan con menor frecuencia en el almacenamiento secundario en disco que en el primario. Es por esto que llamamos a los discos —y a otros dispositivos de almacenamiento secundario— **almacenamiento no volátil**, en tanto que la memoria principal suele caracterizarse como **almacenamiento volátil**.
- El costo de almacenamiento por unidad de información es un orden de magnitud menor en el caso de discos que en el del almacenamiento primario.

Están surgiendo nuevas tecnologías, como el almacenamiento en disco óptico, memorias principales más grandes y económicas, y hardware de propósito especial orientado a las bases de datos. Es posible que en el futuro tales tecnologías ofrezcan alternativas viables al empleo de discos magnéticos, pero por ahora es importante estudiar y comprender las propiedades y características de estos discos magnéticos y la forma en que podemos organizar los archivos de datos en el disco a fin de diseñar bases de datos eficaces con un rendimiento aceptable.

Las cintas magnéticas se utilizan a menudo como medio de almacenamiento para respaldar la base de datos porque el almacenamiento en cinta cuesta aún menos que el almacenamiento en disco. Sin embargo, el acceso a los datos en cinta es muy lento. Los datos

almacenados en cinta están **fuera de línea**; esto es, se requiere la intervención de un operador (o de un dispositivo de carga) para cargar una cinta y poder tener acceso a los datos. En contraste, los discos son dispositivos **en línea** a los cuales se puede tener acceso en cualquier momento.

En este capítulo y el siguiente describiremos las técnicas para almacenar grandes cantidades de datos estructurados en un disco. Estas técnicas son importantes para los diseñadores de bases de datos, para el DBA y para quienes implementan los SGBD. Los diseñadores de bases de datos y el DBA deben conocer las ventajas y desventajas de las técnicas de almacenamiento para diseñar, implementar y operar una base de datos en un SGBD específico. Es común que el SGBD ofrezca varias opciones para organizar los datos, y el proceso de **diseño físico de bases de datos** implica elegir, de entre las opciones disponibles, las técnicas de organización de datos idóneas para los requerimientos de aplicación dados. Los implementadores de SGBD deben estudiar las técnicas de organización de los datos para poder implementarlas de manera eficiente y así ofrecer al DBA y a los usuarios del SGBD suficientes opciones.

En general, las aplicaciones de base de datos sólo requieren una pequeña porción de la base de datos en un momento dado, a fin de procesarla. Siempre que se requiera una cierta porción de los datos, habrá que localizarla en el disco, copiarla en la memoria principal para procesarla y luego escribirla otra vez en el disco si es que se modificaron los datos. Los datos almacenados en el disco están organizados en **archivos** de **registros**. Cada registro es una colección de valores de datos que se pueden interpretar como hechos en torno a entidades, sus atributos y sus vínculos. Los registros deben almacenarse en disco de manera tal que sea posible localizarlos de manera eficiente cuando se les requiera.

Hay varias **organizaciones primarias de archivos** que determinan la forma en que los registros de un archivo *se colocan físicamente en el disco*. Los *archivos de montículo* (o *archivos no ordenados*) colocan los registros en disco sin un orden específico, en tanto que los *archivos ordenados* (o *archivos secuenciales*) mantienen los registros ordenados según el valor de un cierto campo. Los *archivos dispersos* utilizan una función de dispersión para determinar la colocación de los registros en el disco. Otras organizaciones primarias de los archivos, como los *árboles B*, se valen de estructuras de árbol. Estudiaremos las organizaciones primarias de los archivos en las secciones 4.6 a 4.9

4.2 Dispositivos de almacenamiento secundario

En esta sección describiremos algunas de las características de los dispositivos de almacenamiento en disco y cinta magnéticos. Los lectores que ya hayan estudiado esto pueden limitarse a hacer una revisión superficial de esta sección.

4.2.1 *Descripción del hardware de los dispositivos de disco*

Los discos magnéticos sirven para almacenar grandes cantidades de datos. La unidad más básica de almacenamiento en el disco es un solo **bit** de información. Si magnetizamos un área del disco de cierta manera, podemos hacer que represente un valor de bit de 0 (cero) o bien de 1 (uno). Para codificar la información, los bits se agrupan en **bytes** (o **caracteres**). Los bytes suelen tener entre 4 y 8 bits, dependiendo del computador y del dispositivo. Supondremos

que un carácter se almacena en un solo byte, y utilizaremos indistintamente los términos *byte* y *carácter*. La **capacidad** de un disco es el número de bytes que puede almacenar, y suele ser bastante grande. Citaremos las capacidades de los discos en kilobytes (Kbyte o aproximadamente 1000 bytes), megabytes (Mbyte o aproximadamente un millón de bytes) y gigabytes (Gbyte o aproximadamente mil millones de bytes). Los pequeños discos flexibles que se utilizan en los microcomputadores suelen contener entre 400 Kbytes y 1.2 Mbytes; los discos duros de las micros suelen tener capacidades de entre 30 y 250 Mbytes, y los grandes paquetes de discos empleados con minicomputadores y macrocomputadores tienen capacidades que llegan a unos cuantos Gbytes. La capacidad de los discos está aumentando continuamente conforme mejoran las tecnologías.

Sea cual sea su capacidad, todos los discos están hechos de material magnético en forma de un disco circular delgado (Fig. 4.1(a)) y protegidos por una cubierta de plástico o acrílico. Los discos son **de un solo lado** si sólo almacenan información en una de sus superficies, y **de doble lado** si se utilizan las dos superficies. A fin de aumentar la capacidad de almacenamiento los discos se construyen en un **paquete de discos** (Fig. 4.1(b)) que puede incluir hasta 30 superficies. La información se almacena en la superficie del disco en círculos concéntricos *muy angostos*, cada uno de los cuales tiene un diámetro distinto y recibe el

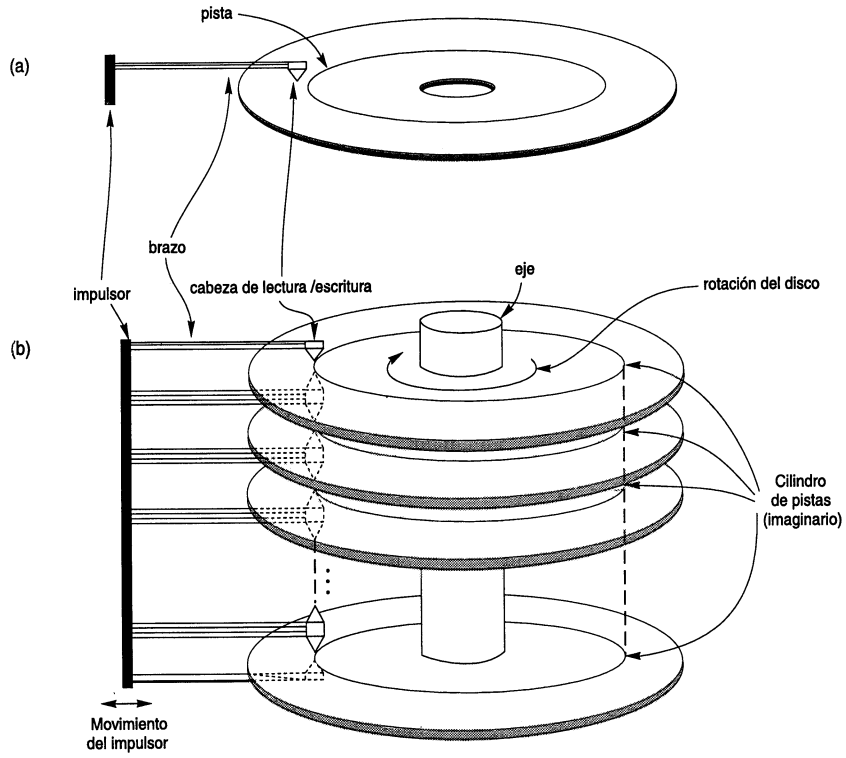


Figura 4.1 (a) Disco de un solo lado con hardware de lectura/escritura.
(b) Paquete de discos con hardware de lectura/escritura.

nombre de **pista**. En el caso de los paquetes de discos, las pistas del mismo diámetro en las diversas superficies constituyen un **cilindro**, por la forma que tendrían si se les conectara en el espacio. El concepto de cilindro es importante porque los datos almacenados en el mismo cilindro se pueden leer con mucha mayor rapidez que si estuvieran distribuidos entre distintos cilindros.

Por lo regular, todos los círculos concéntricos pueden contener la misma cantidad de información, así que los bits están empacados más densamente en las pistas de menor diámetro. El número de pistas de un disco puede llegar hasta 800, y la capacidad de cada pista suele ser de entre 4 y 50 Kbytes. Dado que una pista puede contener una gran cantidad de información, se le divide en bloques más pequeños o sectores. La división de una pista en **sectores** está codificada permanentemente en la superficie del disco y no se puede modificar. Los sectores subtienden un ángulo fijo en el centro (Fig. 4.2), y no todos los discos tienen sus pistas divididas en sectores. La división de una pista en **bloques** de igual tamaño, o *páginas*, la establece el sistema operativo durante la grabación del **formato** (o **iniciación**) del disco. El tamaño de los bloques se fija durante la iniciación, y no es posible alterarlo dinámicamente. Los bloques suelen tener entre 512 y 4096 bytes. En el caso de los discos con sectores codificados permanentemente, es común que los sectores se subdividan en bloques durante la iniciación. La división de los bloques se establece mediante **separaciones entre bloques** de tamaño fijo, que incluyen información de control especialmente codificada que se graba durante la iniciación del disco. Esta información sirve para determinar cuál bloque de la pista sigue a cada separación entre bloques.

Se dice que los discos son dispositivos direccionables *de acceso aleatorio*. La transferencia de los datos entre la memoria principal y el disco se efectúa en unidades de bloques. La **dirección de hardware** de un bloque —una combinación de número de superficie, número de pista (dentro de la superficie) y número de bloque (dentro de la pista)— se proporciona al hardware de entrada/salida (E/S) del disco. También se proporciona la dirección de un **almacenamiento intermedio** (*buffer*): un área reservada contigua en la memoria principal en la que cabe un bloque. En el caso de una orden de **lectura**, el bloque de disco se copia en la memoria intermedia; si la orden es de **escritura**, el contenido de la memoria intermedia se copia en el bloque del disco. Hay ocasiones en las que se transfieren varios bloques contiguos (un **grupo**) como una sola unidad. En este caso el tamaño de la memoria intermedia se ajusta de modo que coincida con el número de bytes del grupo.

De hecho, el mecanismo encargado de leer o escribir bloques es la **cabeza de lectura/escritura** del disco, que forma parte de un sistema llamado **unidad de disco**. Los discos o paquetes de discos se montan en la unidad de disco, que cuenta con un motor para hacer girar los discos. La cabeza de lectura/escritura contiene un componente electrónico unido a

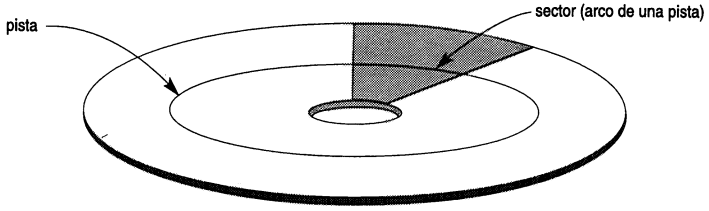


Figura 4.2 Grupo de sectores que subtienden el mismo ángulo.

un **brazo mecánico**. Los paquetes de discos con múltiples superficies tienen también varias cabezas de lectura/escritura, una por cada superficie (Fig. 4.1(b)). Todos los brazos están sujetos a un **impulsor** conectado a otro motor eléctrico, el cual mueve las cabezas de lectura/escritura al unísono y las coloca con precisión sobre el cilindro de pistas especificado en una dirección de bloque.

Las unidades de discos duros giran el paquete de discos continuamente a velocidad constante. En el caso de los discos flexibles, la unidad de disco comienza a girar el disco cada vez que se emite una solicitud de lectura o escritura específica, y deja de girarlo poco después de completarse la transferencia de los datos. Una vez que la cabeza de lectura/escritura está colocada en la pista correcta y que el bloque especificado en la dirección de bloque pasa por la cabeza, el componente electrónico de ésta se activa para transferir los datos. Algunas unidades de disco tienen cabezas de lectura/escritura fijas, con tantas cabezas como pistas tiene el disco. Éstas se denominan discos de **cabeza fija**, en tanto que las unidades provistas de impulsor se llaman discos de **cabeza móvil**. En el caso de los discos de cabeza fija, la pista o el cilindro se escoge electrónicamente conmutando a la cabeza de lectura/escritura apropiada, sin que haya movimiento mecánico; en consecuencia, son mucho más rápidos. Sin embargo, el costo de las cabezas adicionales es bastante alto, y por ello los discos de cabeza fija no suelen utilizarse mucho.

Para transferir un bloque de disco, dada su dirección, la unidad de disco debe ubicar primero mecánicamente la cabeza de lectura/escritura sobre la pista correcta. El tiempo requerido para hacer esto se denomina **tiempo de búsqueda**. Después, hay otro lapso de espera —el **retardo rotacional** o **latencia**— mientras el principio del bloque deseado gira hasta colocarse bajo la cabeza de lectura/escritura. Por último, se requiere un lapso adicional para transferir los datos: el **tiempo de transferencia de bloque**. Por tanto, el tiempo total necesario para localizar y transferir un bloque arbitrario, dada su dirección, es la suma del tiempo de búsqueda, el retardo rotacional y el tiempo de transferencia de bloque. El tiempo de búsqueda y el retardo rotacional suelen ser mucho mayores que el tiempo de transferencia de bloque. Para hacer más eficiente la transferencia de múltiples bloques, se acostumbra transferir varios bloques consecutivos de la misma pista o cilindro. Esto elimina el tiempo de búsqueda y el retardo rotacional para todos los bloques, con excepción del primero, lo que da pie a un ahorro sustancial de tiempo cuando se transfieren muchos bloques contiguos. Por lo regular, el fabricante del disco indica una **velocidad de transferencia masiva** para calcular el tiempo que requiere la transferencia de bloques consecutivos. En el apéndice B hay un análisis sobre éstos y otros parámetros de disco.

El tiempo requerido para localizar y transferir un bloque de disco es del orden de milisegundos, por lo regular entre 15 y 60 ms. En el caso de bloques contiguos, la localización del primer bloque tarda entre 15 y 60 ms, pero la transferencia de los bloques subsecuentes podría tardar sólo 1 o 2 ms por bloque. Muchas técnicas de búsqueda aprovechan la obtención consecutiva de bloques al buscar datos en disco. En todo caso, un tiempo de transferencia del orden de milisegundos se considera bastante alto en comparación con el tiempo necesario para procesar los datos en la memoria principal con las UCP actuales. Por tanto, la localización de datos en el disco es un *cuello de botella importante* en las aplicaciones de bases de datos. Las estructuras de archivos que veremos aquí y en el capítulo 5 intentan *minimizar el número de transferencias de bloques* necesarias para localizar y transferir los datos requeridos del disco a la memoria principal.

4.2.2 *Dispositivos de almacenamiento en cinta magnética*

Los discos son dispositivos de almacenamiento secundario de **acceso aleatorio**, porque es posible tener acceso "de manera aleatoria" a un bloque arbitrario en el disco con sólo especificar su dirección. Las cintas magnéticas son dispositivos de **acceso secuencial**; si queremos tener acceso al n-ésimo bloque de la cinta, tendremos que leer antes los $n - 1$ bloques precedentes. Los datos se almacenan en carretes de cinta magnética de alta capacidad, un tanto parecidos a las cintas de audio o vídeo. Se requiere una **unidad de cinta** para leer los datos de un **carrete de cinta**, o grabarlos en ella. Por lo regular, cada grupo de bits que forman un byte se almacena a lo ancho de la cinta, y los bytes en sí se almacenan consecutivamente en ella.

Los datos se leen o escriben en la cinta mediante una cabeza de lectura/escritura. Los registros de datos también se almacenan en bloques en la cinta, aunque el tamaño de éstos puede ser bastante mayor que en el caso de los discos, y las separaciones entre bloques son también muy grandes. Dadas las densidades comunes de la cinta —entre 4000 y 16 000 bytes por centímetro— una separación entre bloques† usual de 1.52 cm corresponde a entre 960 y 3750 bytes de espacio de almacenamiento desperdiciado. Con el fin de aprovechar mejor el espacio, se acostumbra agrupar muchos registros en un solo bloque.

La característica principal de una cinta es el requisito de tener acceso a los bloques de datos en **orden secuencial**. Si queremos llegar a un bloque ubicado a la mitad de un carrete de cinta, deberemos montar el carrete y leer la cinta hasta que el bloque deseado pase por la cabeza de lectura/escritura. Por esta razón, el acceso en cinta puede ser lento y las cintas no se utilizan para almacenar datos en línea, con excepción de algunas aplicaciones especializadas. Sin embargo, las cintas cumplen con una función muy importante, la de **respaldar** la base de datos. Una razón para hacer este respaldo es mantener copias de seguridad de los archivos de disco por si acaso se pierden los datos debido a un aterrizaje de la cabeza, que ocurre cuando la cabeza de lectura/escritura del disco toca la superficie de este último a causa de una descompostura mecánica. Por esta razón, los archivos del disco se copian periódicamente en cintas. Las cintas también pueden servir para almacenar archivos de datos demasiado grandes. Por último, los archivos de base de datos que se utilizan con muy poca frecuencia, o que carecen de actualidad pero se deben conservar como registros históricos, se pueden **archivar** en cinta. En fechas recientes, las cintas magnéticas más pequeñas (de 8 mm, similares a las que se utilizan en las cámaras-grabadoras de vídeo) y los CD-ROM (discos compactos con memoria sólo de lectura) se han vuelto muy populares como medios para respaldar archivos de datos de estaciones de trabajo y computadores personales, y para almacenar imágenes y bibliotecas del sistema.

4.3 Almacenamiento intermedio de bloques

Cuando es preciso transferir varios bloques del disco a la memoria principal y se conocen por anticipado todas las direcciones de bloque, es posible reservar varias áreas de almacenamiento intermedio (*buffers*) en la memoria principal para agilizar la transferencia. Mientras se lee o escribe un *buffer*, la UCP puede procesar los datos de otro. Esto es posible porque casi siempre se cuenta con un procesador de entrada/salida (E/S) de disco independiente que, una

†Llamada *separación entre registros* en la terminología de cintas.

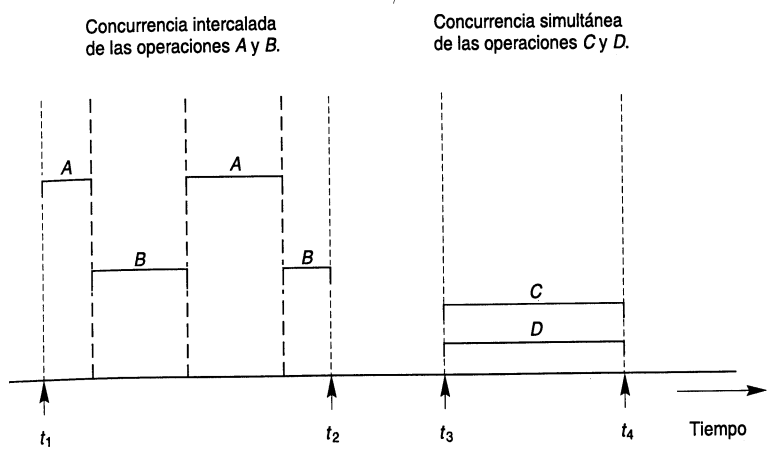


Figura 4.3 Concurrencia intercalada y simultánea.

vez activado, puede proceder a transferir un bloque de datos entre la memoria y el disco, independientemente del proceso de la UCP y en paralelo con él.

La figura 4.3 ilustra cómo dos procesos pueden efectuarse en paralelo. Los procesos A y B se ejecutan de manera **concurrente** en forma **intercalada**, en tanto que los procesos C y D se ejecutan en forma **concurrente** de manera **simultánea**. Cuando una sola UCP controla múltiples procesos, la ejecución simultánea no es posible. No obstante, los procesos se pueden ejecutar concurrentemente de manera intercalada. La mayor utilidad de la memoria intermedia se da cuando los procesos pueden ejecutarse concurrentemente de manera simultánea, sea porque se cuenta con un procesador de E/S de disco aparte, o porque el computador tiene múltiples procesadores.

La figura 4.4 ilustra la manera en que la lectura y el procesamiento pueden efectuarse en paralelo cuando el tiempo necesario para procesar un bloque de disco en la memoria es menor que el tiempo necesario para leer el siguiente bloque y llenar un *buffer*. La UCP puede comenzar a procesar un bloque una vez completada su transferencia a la memoria principal; al mismo tiempo, el procesador de E/S de disco puede estar leyendo y transfiriendo el siguiente bloque a un almacenamiento intermedio diferente. Esta técnica se conoce como **doble almacenamiento intermedio** y también puede servir para escribir un flujo continuo de bloques de la memoria en el disco. El doble almacenamiento intermedio permite la lectura o escritura continua de datos en bloques consecutivos del disco, eliminando así el tiempo de búsqueda y el retardo rotacional de todas las transferencias de bloque, con excepción de la primera. Por añadidura, los datos están listos para procesarse, lo que reduce el tiempo de espera en los programas.

4.4 Grabación de registros de archivo en disco

En esta sección definiremos los conceptos de registro, tipo de registros y archivo. En seguida, analizaremos diferentes técnicas para colocar los registros de un archivo en el disco.

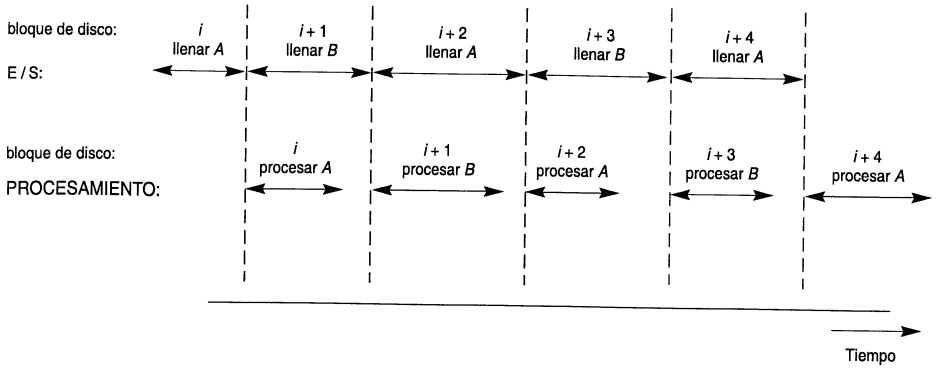


Figura 4.4 Empleo de dos áreas de memoria intermedia, *A* y *B*, para leer del disco.

4.4.1 *Tipos de registros*

Los datos casi siempre se almacenan en forma de **registros**. Cada registro consiste en una colección de **valores** o **elementos** de información relacionados, donde cada valor se forma de uno o más bytes y corresponde a un determinado **campo** del registro. En general, los registros describen entidades y sus atributos. Por ejemplo, un registro EMPLEADO representa una entidad empleado, y cada valor de campo del registro especifica un atributo de ese empleado, como NOMBRE, FECHA-NACIMIENTO, SALARIO o SUPERVISOR. Una colección de nombres de campos y sus tipos de datos correspondientes constituye una definición de **tipo de registro** o **formato de registro**. El **tipo de datos** asociado a cada campo especifica el tipo de valores que el campo puede aceptar.

El tipo de datos de un campo casi siempre es uno de los tipos de datos estándar empleados en programación. Entre ellos se cuentan los tipos de datos numéricos (entero, entero largo o número real), de cadena de caracteres (longitud fija o variable), booleanos (que sólo adoptan los valores 0 y 1 o FALSO y VERDADERO), y a veces tipos de **fecha** y **hora** especialmente codificados. El número de bytes requerido para cada tipo de datos es fijo para un computador en particular. Un entero podría requerir 4 bytes, un entero largo, 8 bytes, un número real, 4 bytes, un booleano, 1 byte, una fecha, 4 bytes (para codificar la fecha como entero) y una cadena de longitud fija de *k* caracteres, *k* bytes. Las cadenas de longitud variable pueden requerir tantos bytes como caracteres tengan los valores de cada campo. Por ejemplo, podríamos definir un tipo de registro EMPLEADO —con la notación de PASCAL— como sigue:

```
NOMBRE DEL TIPO DE REGISTRO   NOMBRES DE CAMPOS   TIPOS DE DATOS
type EMPLEADO = record        NOMBRE              :packed array [1..30] of character;
                              NSS                 :packed array [1..9] of character;
                              SALARIO             :integer;
                              CÓDIGO_PUESTO       :integer;
                              DEPARTAMENTO        :packed array [1..20] of character;
                              end;
```


En aplicaciones recientes de base de datos, puede haber necesidad de almacenar elementos de información que consistan en objetos grandes no estructurados, que representen imágenes, flujos de vídeo o audio digitalizados, o texto libre. Estos se denominan **objetos binarios grandes** (**BLOB**: *binary large objects*). Por lo regular, un elemento de información BLOB se almacenará aparte de su registro, en un área de bloques de disco, y se incluirá en el registro un apuntador al BLOB.

4.4.2 *Archivos, registros de longitud fija y registros de longitud variable*

Un **archivo** es una *secuencia* de registros. En muchos casos, todos los registros de un archivo son del mismo tipo. Si todos tienen exactamente el mismo tamaño (en bytes), se dice que el archivo se compone de registros de **longitud fija**. Si diferentes registros del archivo tienen tamaños distintos, se dice que el archivo está constituido por registros de **longitud variable**. Un archivo puede tener registros de longitud variable por varias razones:

- Los registros del archivo son todos del mismo tipo, pero uno o más de los campos son de tamaño variable (**campos de longitud variable**). Por ejemplo, el campo NOMBRE de EMPLEADO puede ser un campo de longitud variable.
- Los registros del archivo son todos del mismo tipo, pero uno o más de los campos pueden tener múltiples valores en registros individuales; se dice que un campo así es un **campo repetitivo**, y el grupo de valores del campo se conoce como **grupo repetitivo**.
- Los registros del archivo son todos del mismo tipo, pero uno o más de los campos son **opcionales**; esto es, pueden tener valores en algunos de los registros, pero quizá no en todos (**campos opcionales**).
- El archivo contiene registros de *diferentes tipos* y por tanto de tamaño variable (**archivo mixto**). Esto ocurriría si se colocaran registros relacionados de diferentes tipos juntos en los bloques de disco; por ejemplo, los registros BOLETA_DE_NOTAS de un estudiante dado se podrían colocar inmediatamente después del registro de ese ESTUDIANTE.

Los registros EMPLEADO de longitud fija de la figura 4.5(a) tienen un tamaño de 71 bytes. Todos los registros tienen los mismos campos, y las longitudes de los campos son fijas para poder identificar la posición inicial de cada campo en relación con la posición inicial del registro. Esto facilita la localización de los valores de los campos con los programas que tienen acceso a tales archivos. Cabe señalar que es posible representar como archivo de registros de longitud fija un archivo que lógicamente debería tener registros de longitud variable. Por ejemplo, en el caso de los campos opcionales podríamos incluir *todos los campos* en *todos los registros*, pero almacenar un valor nulo especial si no existe valor para ese campo. En el caso de un campo repetitivo, podríamos asignar a cada registro el máximo de bytes que puedan ocupar los valores de ese campo. En ambos casos se desperdicia espacio si algunos registros no tienen valores para todos los espacios físicos disponibles en cada registro. En seguida veremos otras opciones para dar formato a los registros de un archivo de registros de longitud variable.

En el caso de los *campos de longitud variable*, todos los registros tienen un valor en cada campo, pero no conocemos la longitud exacta de los valores de algunos campos. Para determinar los bytes que representan cada campo dentro de un registro en particular, podemos

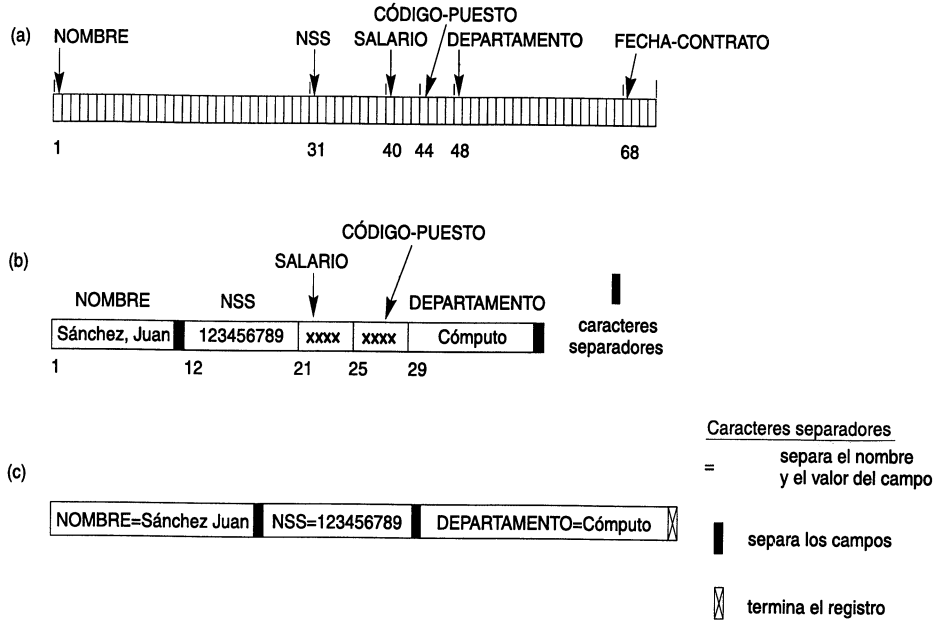


Figura 4.5 Diversos formatos de almacenamiento de registros. (a) Registro de longitud fija con seis campos y 71 bytes de longitud. (b) Registro con dos campos de longitud variable y tres de longitud fija. (c) Un registro de campos variables con tres tipos de caracteres separadores.

usar caracteres **separadores** especiales, que no aparecen en ningún valor de campo (como ? o % o $), para terminar los campos de longitud variable (Fig. 4.5 (b)), o bien podemos almacenar la longitud de cada campo del registro.

Los archivos de registros con *campos opcionales* pueden tener diferentes formatos. Si el número total de campos del tipo de registros es grande, pero el número de campos que aparecen realmente en un registro representativo es pequeño, podemos incluir en cada campo una secuencia de pares <nombre-de-campo, valor-de-campo> y no sólo los valores de los campos. En la figura 4.5(c) se utilizan tres tipos de caracteres separadores, aunque podríamos usar el mismo carácter separador para los dos primeros propósitos: separar el nombre del campo y su valor, y separar un campo del siguiente. Una opción más práctica es asignar un **tipo de campo** corto —digamos, un número entero— a cada campo e incluir en cada registro una secuencia de pares <tipo-de-campo, valor-de-campo> en vez de pares <nombre-de-campo, valor-de-campo>.

Los *campos repetitivos* necesitan un carácter separador para separar los valores repetitivos del campo y otro para indicar la terminación del campo. Por último, en el caso de archivos que contengan *registros de diferentes tipos*, cada registro va precedido por un indicador **de tipo de registro**. Obviamente, los programas que procesan registros de longitud variable tendrán que ser más complejos que los que manejan registros de longitud fija, donde se conocen la posición inicial y el tamaño de cada campo, y nunca cambian.

4.4.3 *Grabación de registros en bloques y registros extendidos vs. no extendidos*

Los registros de un archivo se deben asignar a bloques del disco porque el bloque es la *unidad de transferencia de datos* entre el disco y la memoria. Si el tamaño del bloque es mayor que el del registro, cada bloque contendrá más de un registro. Algunos archivos pueden tener registros desusadamente grandes que no caben en un bloque. Supongamos que el tamaño de bloque es B bytes. Para un archivo de registros de longitud fija de tamaño R, con $B \geq R$, podemos colocar $fbl = \lfloor (B/R) \rfloor$ registros por bloque, donde $\lfloor (x) \rfloor$ (*función piso*) redondea el valor x hacia abajo hasta el siguiente entero. El valor fbl se denomina **factor de bloques** del archivo. En general, R no será divisor exacto de B, de modo que tendremos cierto espacio desocupado en cada bloque igual a

$$B - (fbl * R) \text{ bytes}$$

Para aprovechar este espacio no utilizado, podemos almacenar parte de un registro en un bloque y el resto en otro. Un **apuntador** al final del primer bloque apuntará al bloque que contiene el resto del registro en caso de que no sea el siguiente bloque consecutivo del disco. Esta organización se llama **extendida**, porque los registros pueden extenderse más allá del final de un bloque. Siempre que un registro sea mayor que un bloque, *es necesario* emplear una organización extendida. Si no se permite que los registros crucen los límites de los bloques, se dice que la organización es **no extendida**. Ésta se usa con registros de longitud fija en los que $B \geq R$, porque hace que cada registro comience en una posición conocida del bloque, lo cual simplifica el procesamiento de registros. En el caso de registros de longitud variable, se puede usar una organización extendida o no extendida. Si el tamaño medio de los registros es grande, resultará ventajoso emplear la organización extendida para reducir el espacio perdido en cada bloque. La figura 4.6 ilustra la organización extendida comparándola con la no extendida.

En el caso de registros de longitud variable que utilizan la organización extendida, cada bloque puede almacenar un número distinto de registros del archivo. Aquí, el factor de bloques fbl representa el número *medio* de registros por bloque para el archivo. Podemos usar fbl para calcular el número de bloques que se necesitan para un archivo de r registros:

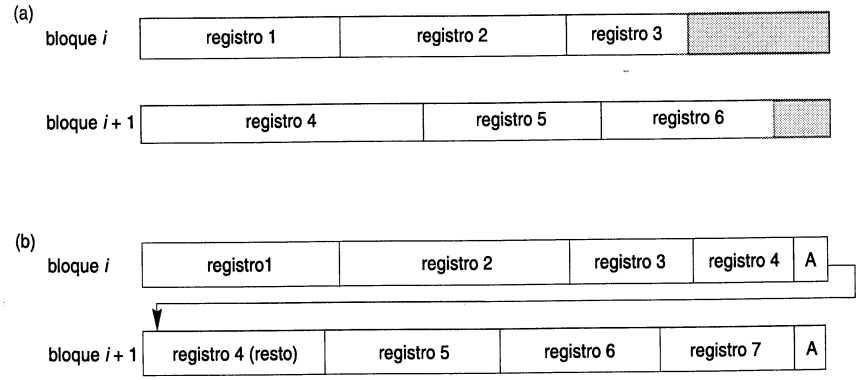


Figura 4.6 Tipos de organización de registros. (a) No extendida. (b) Extendida.

$$b = \lceil (r/fbl) \rceil \text{ bloques}$$

donde $\lceil (x) \rceil$ (*función techo*) redondea el valor de x hacia arriba hasta el siguiente entero.

4.4.4 *Asignación de bloques de archivo en disco*

Existen varias técnicas estándar para asignar los bloques de un archivo en disco. En la **asignación contigua** los bloques del archivo se asignan a bloques consecutivos del disco. Esto agiliza notablemente la lectura de todo el archivo si se emplea doble memoria intermedia, pero dificulta la expansión del archivo. En la **asignación enlazada** cada bloque del archivo contiene un apuntador al siguiente bloque de ese archivo. Esto facilita la expansión del archivo pero vuelve más lenta su lectura. Una combinación de las dos asigna **grupos** de bloques de disco consecutivos, y luego enlaza los grupos. A estos grupos se les llama en ocasiones **segmentos** o **alcances**. Otra posibilidad es utilizar la **asignación indizada**, donde uno o más **bloques de índice** contienen apuntadores a los bloques de archivo reales. También es frecuente el empleo de combinaciones de estas técnicas.

4.4.5 *Descriptores de archivo*

Un **descriptor de archivo** o **cabecera de archivo** contiene información relativa al archivo que es necesaria para los programas que tienen acceso a los registros de dicho archivo. El descriptor contiene información para determinar las direcciones en disco de los bloques del archivo, y también para registrar descripciones de formato, como las longitudes de los campos y el orden de los campos dentro de un registro en el caso de los registros no extendidos de longitud fija, y códigos de tipo de campo, caracteres separadores y códigos de tipo de registro en el caso de los registros de longitud variable.

Para buscar un registro en el disco, se copian uno o más bloques en almacenamiento intermedio de la memoria principal. En seguida, los programas buscan el registro o registros deseados dentro del almacenamiento intermedio, utilizando la información del descriptor del archivo. Si se desconoce la dirección del bloque que contiene el registro deseado, los programas de búsqueda deberán efectuar una **búsqueda lineal** a través de los bloques del archivo. Se copia cada bloque en un *buffer* y se examina hasta encontrar el registro o hasta haber buscado en todos los bloques del archivo infructuosamente. Esto puede requerir mucho tiempo si el archivo es grande. El objetivo de una buena organización de archivo es localizar el bloque que contiene un registro dado con un mínimo de transferencias de bloques.

4.5 Operaciones con archivos

Las operaciones con archivos suelen dividirse en operaciones de **obtención** y operaciones de **actualización**. Las primeras no alteran los datos del archivo, pues sólo localizan ciertos registros para poder examinar los valores de sus campos y procesarlos. Las segundas modifican el archivo por la inserción o eliminación de registros o por la modificación de los valores de los campos. En ambos casos, tal vez tengamos que **seleccionar** uno o más registros para su obtención, eliminación o modificación con base en una **condición de selección**, que especifica los criterios que el registro o registros deseados deben satisfacer.

Consideremos un archivo EMPLEADO con los campos NOMBRE, NSS, SALARIO, CÓDIGO_PUESTO y DEPARTAMENTO. Una **condición de selección simple** podría implicar una comparación de igualdad de algún valor de campo; por ejemplo, (NSS = '123456789') o (DEPARTAMENTO = 'Investigación'). Y condiciones más complejas pueden implicar otros tipos de operadores de comparación, como > o ≥; un ejemplo es (SALARIO ≥ 30 000). El caso general es tener como condición de selección una expresión booleana arbitraria para los campos del archivo.

Las operaciones de búsqueda en los sistemas de archivos se basan generalmente en condiciones de selección simples. Si la condición es compleja, el SGBD (o el programador) deberá descomponerla para extraer una condición simple que pueda servir para localizar los registros en el disco. A continuación se revisa cada uno de los registros localizados para determinar si satisfacen la condición de selección completa. Por ejemplo, podemos extraer la condición simple (DEPARTAMENTO = 'Investigación') de la condición compleja ((SALARIO ≥ 30 000) Y (DEPARTAMENTO = 'Investigación')); localizaremos todos los registros que satisfagan (DEPARTAMENTO = 'Investigación') y veremos si también satisfacen (SALARIO ≥ 30 000).

Cuando varios registros del archivo satisfacen una condición de búsqueda, sólo se localiza el *primer* registro (con respecto a la secuencia física de los registros en el archivo). Para localizar los demás registros que satisfacen la condición, es preciso efectuar operaciones adicionales. El registro localizado más recientemente se designa **registro actual**. Las operaciones de búsqueda subsecuentes parten de este registro y localizan el *siguiente* que satisface la condición.

Las operaciones reales para localizar y leer los registros de un archivo varían de un sistema a otro. A continuación presentamos un conjunto de operaciones representativas. Por lo regular, los programas de alto nivel, como el software de SGBD, tienen acceso a los registros valiéndose de estas órdenes; es por ello que en ocasiones mencionaremos **variables del programa** en la siguiente descripción:

- *Buscar* (*Find*) o *Localizar* (*Locate*): Busca el primer registro que satisface una condición de búsqueda. Transfiere el bloque que contiene ese registro a un *buffer* en la memoria principal (si es que no está ahí ya). El registro se localiza en el *buffer* y se convierte en el registro actual. A veces se utilizan diferentes verbos para indicar si el registro localizado sólo se va a obtener (Buscar) o también se va a actualizar (Localizar).
- *Leer* (*Read*) u *Obtener* (*Get*): Copia el registro actual del almacenamiento intermedio a una variable del programa o a un área de trabajo del programa del usuario. Esta orden también puede avanzar el apuntador del registro actual al siguiente registro del archivo, para lo cual puede ser necesario transferir del disco el siguiente bloque del archivo.
- *BuscarSiguiente* (*FindNext*): Busca en el archivo el siguiente registro que satisface la condición de búsqueda. Transfiere el bloque que contiene ese registro a un *buffer* en la memoria principal (si es que no está ahí ya). El registro se localiza en el *buffer* y se convierte en el registro actual.
- *Eliminar* (*Delete*): Elimina el registro actual y (tarde o temprano) actualiza el archivo en el disco de modo que refleje la eliminación.
- *Modificar* (*Modify*): Modifica algunos valores de campos del registro actual y (tarde o temprano) actualiza el archivo en el disco de modo que refleje la modificación.

- *Insertar* (*Insert*): Inserta un registro nuevo en el archivo según los pasos siguientes: localiza el bloque donde se debe insertar el registro, transfiere dicho bloque a un área de almacenamiento intermedio (si es que no está ahí ya), escribe el registro en esa área y (tarde o temprano) escribe el contenido del *buffer* en el disco para que el archivo refleje la inserción.

Las operaciones anteriores se denominan **de registro por registro** porque cada una se aplica a un solo registro. En algunos sistemas de archivos es posible aplicar operaciones adicionales de más alto nivel, **de conjunto por conjunto**. Como ejemplos podemos citar las siguientes:

- *BuscarTodos* (*FindAll*): Localiza *todos* los registros del archivo que satisfacen una condición de búsqueda.
- *BuscarOrdenados* (*FindOrdered*): Obtiene todos los registros del archivo en algún orden especificado.
- *Reorganizar* (*Reorganize*): Inicia el proceso de reorganización. Como veremos, algunas organizaciones de archivos requieren una reorganización periódica. Un ejemplo sería reordenar los registros del archivo de acuerdo con un cierto campo.

Se requieren otras operaciones que preparan el archivo para el acceso (**Abrir**, *Open*) y que indican que ya terminamos de usar el archivo (**Cerrar**, *Close*). La operación Abrir casi siempre lee el descriptor del archivo y prepara el almacenamiento intermedio para las operaciones subsecuentes con el archivo.

En este punto, vale la pena señalar la diferencia entre los términos *organización del archivo* y *método de acceso*. La **organización del archivo** se refiere a la organización de los datos de un archivo en registros, bloques y estructuras de acceso; esto incluye la forma en que los registros y los bloques se colocan en el medio de almacenamiento y se interconectan. El **método de acceso**, en cambio, consiste en un grupo de programas que permiten la aplicación de operaciones (como las que listamos antes) a un archivo. En general, es posible aplicar varios métodos de acceso diferentes a una organización de archivo, aunque algunos de estos métodos sólo pueden aplicarse a archivos que están organizados de cierta manera. Por ejemplo, no podemos aplicar un método de acceso indizado a un archivo que carece de índice (véase el Cap. 5).

Por lo regular, es de esperarse que utilizaremos algunas condiciones de búsqueda más que otras. Algunos archivos pueden ser **estáticos**, lo que significa que pocas veces se efectúan operaciones de actualización; otros archivos, más **volátiles**, tal vez cambien con frecuencia, lo que significa que se les aplican constantemente operaciones de actualización. Una organización de archivo idónea deberá realizar de la manera más eficiente posible las operaciones que esperamos *aplicar a menudo* al archivo. Por ejemplo, consideremos el archivo EMPLEADO antes descrito, que almacena los registros de los empleados actuales de una compañía. Esperaremos insertar registros (al contratar empleados), eliminar registros (cuando los empleados abandonen la empresa) y modificar registros (por ejemplo, cuando se cambia el salario de un empleado). La eliminación o modificación de un registro requiere una condición de selección para identificar un registro o conjunto de registros en particular. La obtención de uno o más registros también requiere una condición de selección.

Si los usuarios esperan aplicar principalmente una condición basada en el NSS, el diseñador deberá elegir una organización de archivo que facilite la localización de un registro dado su valor de NSS. Para ello, tal vez se ordenen físicamente los registros según su valor de NSS, o se defina un índice por NSS (véase el Cap. 5). Suponga que una segunda aplicación utiliza el archivo para generar los cheques de nómina de los empleados y necesita agrupar los cheques por departamento. Para esta aplicación lo mejor es almacenar de manera contigua todos los registros de empleado que tienen el mismo valor de departamento, empacándolos en bloques y quizá ordenándolos por nombre dentro de cada departamento. Sin embargo, esta organización no es compatible con la ordenación de los registros por su valor de NSS. Si es posible, el diseñador deberá elegir una organización que permita efectuar de manera eficiente ambas operaciones. Desafortunadamente, en muchos casos puede ser que no exista ninguna organización que permita implementar con eficiencia todas las operaciones requeridas. Algunas organizaciones de archivo hacen muy eficientes las obtenciones con ciertas condiciones de búsqueda, pero a expensas de hacer muy costosas las actualizaciones. En tales casos, debe elegirse un término medio congruente con la mezcla esperada de operaciones de obtención y de actualización.

En las secciones que siguen y en el capítulo 5 estudiaremos diferentes métodos para organizar los registros de un archivo en el disco. Hay varias técnicas generales, como el ordenamiento, la dispersión y la indización, que sirven para crear métodos de acceso. Por añadidura, hay diversas técnicas generales para efectuar inserciones y eliminaciones que funcionan con muchas organizaciones de archivos.

4.6 Archivos de registros no ordenados (archivos de montículo)

En el tipo más simple y básico de organización, los registros se colocan en el archivo en el orden en que se insertan, y los registros nuevos se insertan al final del archivo. Un archivo con este tipo de organización se denomina archivo de **montículo** o de **montón**.† Esta organización suele utilizarse con caminos de acceso adicionales, como los índices secundarios que trataremos en el capítulo 5. También sirve para reunir y almacenar registros de datos que se utilizarán en el futuro.

La inserción de un registro nuevo es *muy eficiente*: el último bloque del archivo en disco se copia en un *buffer*, se añade el nuevo registro, y se **reescribe** el bloque en el disco. La dirección del último bloque del archivo se guarda en el descriptor del archivo. Por otro lado, buscar un registro con base en cualquier condición de búsqueda requiere una **búsqueda lineal** del archivo, bloque por bloque, que es un procedimiento muy costoso. Si sólo un registro satisface la condición de búsqueda, los programas transferirán a la memoria y examinarán la mitad de los bloques del archivo, en promedio, antes de hallar el registro. En el caso de un archivo de b bloques, esto requiere buscar en $(b/2)$ bloques, en promedio. Si ningún registro satisface la condición de búsqueda, o si varios lo hacen, el programa deberá leer y examinar los b bloques del archivo.

Para eliminar un registro, el programa encargado de ello primero tendrá que hallarlo, copiar el bloque en un *buffer*, eliminar el registro del *buffer* y finalmente **reescribir el bloque** en el disco. Esto deja espacio desocupado adicional en el bloque, así que la eliminación de un gran número de registros de esta forma origina una gran cantidad de espacio desperdiciado.

†Algunos sistemas de archivos como el VAX RMS (Record Management Services, servicios de gestión de registros) de Digital Equipment Corporation llaman a esta organización *archivo secuencial*.

Otra técnica para eliminar registros consiste en tener un byte o bit adicional, el llamado **marcador de eliminación**, almacenado con cada registro. Para eliminar un registro se asigna un cierto valor al marcador de eliminación, así que un valor distinto de este último indica un registro válido (no eliminado). Los programas de búsqueda examinan sólo los registros válidos de un bloque al efectuar su búsqueda. Estas dos técnicas de eliminación requieren una **reorganización** periódica del archivo para recuperar el espacio desocupado que van dejando los registros eliminados. Durante la reorganización, se tiene acceso consecutivo a los bloques del archivo, y los registros se empacan quitando los registros eliminados. Después de la reorganización, los bloques quedan completamente llenos otra vez. Otra posibilidad es aprovechar el espacio de los registros eliminados al insertar registros nuevos, aunque esto requiere cálculos adicionales para seguir la pista de las posiciones vacías.

Con los archivos no ordenados podemos usar una organización extendida o no extendida y registros de longitud fija o variable. La modificación de un registro de longitud variable podría requerir la eliminación del registro antiguo y la inserción del registro modificado, porque es posible que éste no quepa en el espacio que ocupaba antes en el disco.

Para leer todos los registros en orden según los valores de algún campo, se crea una copia ordenada del archivo. Como la ordenación es una operación costosa cuando el archivo ocupa mucho espacio en el disco, se utilizan técnicas especiales de **ordenación externa**. Un método común es una variación de la técnica de ordenación por fusión (*merge-sort*). En primer lugar, se ordenan los registros dentro de cada bloque. Luego se fusionan los bloques ordenados para crear grupos de registros ordenados, cada uno del tamaño de dos bloques. Estos grupos suelen recibir el nombre de **series**; las series de dos bloques se fusionan para formar series de cuatro bloques, y así sucesivamente, hasta que la serie final es el archivo completo ordenado.

En el caso de *registros de longitud fija* no ordenados que utilizan *bloques no extendidos* y *asignación contigua*, es muy sencillo tener acceso a cualquier registro por su **posición** en el archivo. Si asignamos a los registros los números 0, 1, 2, ..., $r - 1$, y a los registros de cada bloque los números 0, 1, ..., $fbl - 1$, donde fbl es el factor de bloques, el i-ésimo registro del archivo se encontrará en el bloque $\lfloor (i/fbl) \rfloor$ y será el (i mod fbl)-ésimo registro de ese bloque. Los archivos de esta especie suelen denominarse **archivos relativos**[†] porque es fácil tener acceso a los registros por sus posiciones relativas. Tener acceso a un registro por su posición no ayuda a localizar un registro con base en una condición de búsqueda, pero sí facilita la construcción de caminos de acceso al archivo, como los índices que trataremos en el capítulo 5.

4.7 Archivos de registros ordenados (archivos ordenados)

Podemos ordenar físicamente los registros de un archivo en disco con base en los valores de uno de sus campos, llamado **campo de ordenación**. Esto da lugar a un archivo **ordenado** o **secuencial**.[††] Si el campo de ordenación también es un **campo clave** del archivo (un campo con un valor único garantizado para cada registro), recibe también el nombre de **clave de ordenación** del archivo. La figura 4.7 muestra un archivo ordenado con NOMBRE como campo clave de ordenación (suponiendo que todos los empleados tienen nombres distintos).

[†]Por ejemplo, VAX RMS (Record Management Services) llama a esta organización *archivo relativo*.

[††]Algunos sistemas de archivos, como el VAX RMS de Digital Equipment Corporation, emplean el término *archivo secuencial* para denotar el archivo no ordenado que describimos en la sección 4.6.

Los archivos ordenados tienen ciertas ventajas sobre los no ordenados. En primer lugar, la lectura de los registros en orden según los valores del campo de ordenación resulta en extremo eficiente, ya que no es necesario ordenarlos adicionalmente. En segundo lugar, encontrar el registro que sigue al actual en orden según el campo de ordenación casi nunca requiere accesos a bloques adicionales, porque el siguiente registro está en el mismo bloque que el actual (a menos que éste sea el último registro del bloque). En tercer lugar, si usamos una condición de búsqueda basada en el valor de un campo clave de ordenación, tendremos acceso más rápido mediante la técnica de búsqueda binaria, lo que constituye una mejora sobre las búsquedas lineales, aunque no se utilice con frecuencia con archivos en disco.

La **búsqueda binaria** en archivos de disco puede efectuarse sobre bloques en vez de sobre registros. Supongamos que el archivo tiene b bloques numerados 1, 2, ..., b; los registros están ordenados por valor ascendente de su campo clave de ordenación; y lo que buscamos es un registro cuyo valor del campo clave de ordenación sea K. Si las direcciones en disco de los bloques están disponibles en el descriptor del archivo, la búsqueda binaria puede describirse con el algoritmo 4.1. Una búsqueda binaria obtiene acceso a $\log_2(b)$ bloques en promedio, sea que se encuentre o no el registro; esto es una mejora con respecto a las búsquedas lineales, donde, en promedio, se obtiene acceso a (b/2) bloques cuando se encuentra el registro y a b bloques cuando no se le encuentra.

ALGORITMO 4.1 Búsqueda binaria sobre una clave de ordenación de un archivo en disco.

```
l ← 1; u ← b; (* b es el número de bloques del archivo *)
mientras (u ≥ l) hacer
  comenzar i ← (l + u) div 2;
  leer bloque i del archivo y colocarlo en almacenamiento intermedio;
  si K < valor del campo clave de ordenación del primer registro del bloque
  entonces u ← i – 1
  si no si K > valor del campo clave de ordenación del último registro del bloque
       entonces l ← i + 1
       si no si el registro con valor del campo clave de ordenación = K está en
  almacenamiento intermedio
            entonces ir a se_encontró
            si no ir a no_se_encontró;
  terminar;
ir a no_se_encontró;
```

Los criterios de búsqueda que comprenden las condiciones >, <, ≥ y ≤ sobre el campo de ordenación son muy eficientes, ya que el ordenamiento físico de los registros implica que todos los registros que satisfacen la condición están contiguos en el archivo. Por ejemplo, y con referencia a la figura 4.7, si el criterio de búsqueda es (NOMBRE < 'F') —donde < significa "alfabéticamente anterior a"—, los registros que satisfacen dicho criterio son los que están entre el principio del archivo y el primer registro cuyo valor de NOMBRE comienza con la letra F.

El ordenamiento no ofrece ninguna ventaja para el acceso aleatorio u ordenado a los registros con base en los valores de un campo *que no sea el de ordenación* del archivo. En tales casos realizaremos una búsqueda lineal para el acceso aleatorio. Si queremos tener acceso en orden a los registros con base en otro campo que no sea el de ordenación, tendremos que crear otra copia ordenada del archivo.

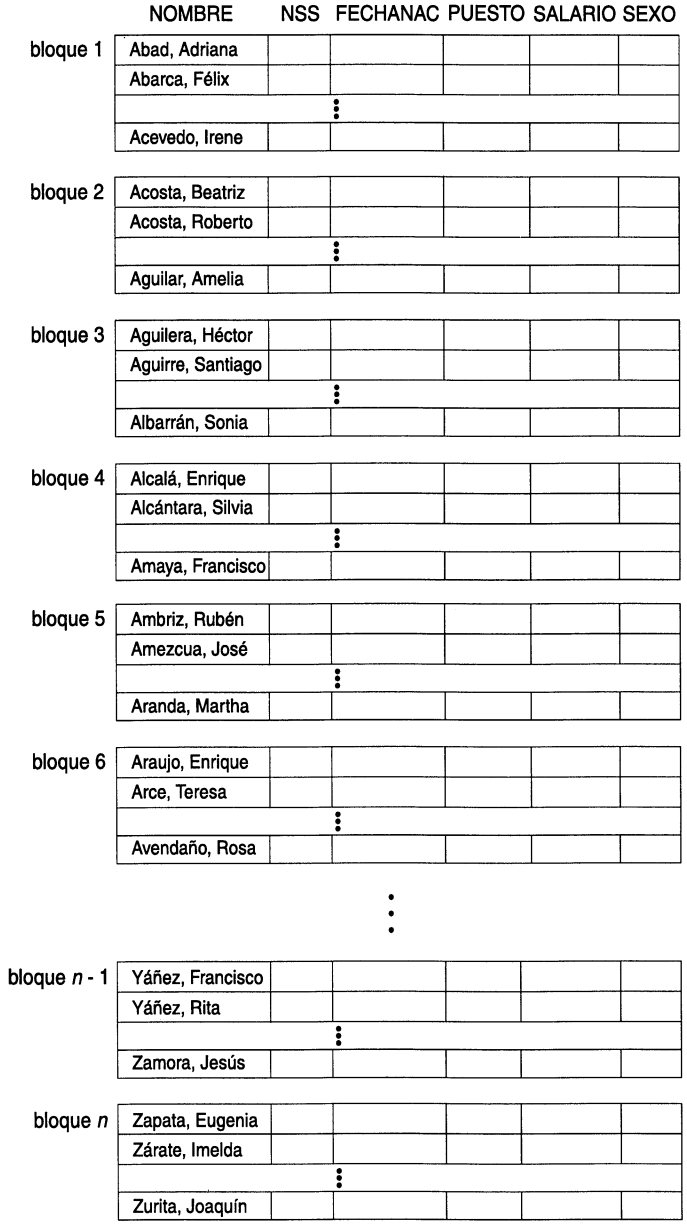


Figura 4.7 Bloques de un archivo ordenado (secuencial) de registros EMPLEADO, con NOMBRE como campo de ordenación.

La inserción y eliminación de registros son operaciones costosas en el caso de archivos ordenados porque se debe conservar el orden de los registros. Para insertar un nuevo registro, tendremos que encontrar su posición correcta en el archivo, con base en su valor del campo de ordenación, y luego abrir espacio en el archivo para colocar el registro en esa posición. Si el archivo es grande, esto puede requerir mucho tiempo, porque será necesario desplazar, en promedio, la mitad de los registros para poder insertar el nuevo. Esto significa que tendremos que leer y reescribir la mitad de los bloques del archivo después de desplazar los registros entre ellos. Para la eliminación de registros, el problema es menos grave si usamos marcadores de eliminación y reorganizamos el archivo periódicamente.

Una opción para hacer más eficiente la inserción es mantener en cada bloque un poco de espacio desocupado para nuevos registros. Sin embargo, una vez agotado este espacio, el problema original reaparecerá. Otro método que se utiliza con frecuencia consiste en crear un archivo *no ordenado* temporal llamado archivo **de desborde** o **de transacciones**. Con esta técnica, el archivo ordenado real se denomina archivo **principal** o **maestro**. Los nuevos registros se insertan al final del archivo de desborde, no en su posición correcta en el archivo principal. Periódicamente, el archivo de desborde se fusiona con el maestro durante la reorganización del archivo. La inserción se vuelve muy eficiente, pero a expensas de un aumento en la complejidad del algoritmo de búsqueda. Será preciso examinar el archivo de desborde con una búsqueda lineal si, después de la búsqueda binaria, no se encuentra el registro en el archivo principal. Si la aplicación no requiere la información más reciente, será posible hacer caso omiso de los registros de desborde durante las búsquedas.

La modificación del valor de un campo de un registro depende de dos factores: la condición de búsqueda para localizar el registro, y el campo por modificar. Si la condición de búsqueda se basa en el campo clave de ordenación, podremos localizar el registro realizando una búsqueda binaria; en caso contrario, tendremos que efectuar una búsqueda lineal. Los campos que no son de ordenación se pueden cambiar modificando el registro y reescribiéndolo en la misma posición física en el disco, siempre que los registros sean de longitud fija. Si el campo modificado es el de ordenación, es probable que el registro deba cambiar de lugar en el archivo, lo que requerirá la eliminación del registro antiguo seguida de la inserción del registro modificado.

La lectura de los registros del archivo en orden según el campo de ordenación es muy eficiente si hacemos caso omiso de los registros de desborde, pues los bloques se pueden leer consecutivamente empleando doble almacenamiento intermedio. Para incluir los registros de desborde, deberemos insertarlos por fusión en sus posiciones correctas; en este caso, podemos reorganizar primero el archivo y luego leer sus bloques en secuencia. Para reorganizar el archivo, lo primero que se hace es ordenar los registros del archivo de desborde, y luego fusionarlos con el archivo maestro. Los registros marcados como eliminados se desechan durante la reorganización.

Los archivos ordenados pocas veces se utilizan en aplicaciones de bases de datos, a menos que se incluya un camino de acceso adicional, llamado *índice primario*, junto con el archivo. Esto mejora todavía más el tiempo de acceso aleatorio sobre el campo clave de ordenación. Hablaremos de los índices en el capítulo 5.

4.8 Técnicas de dispersión

Otro tipo de organización primaria de archivos se basa en la dispersión (*hashing*), que proporciona un acceso muy rápido a los registros con ciertas condiciones de búsqueda. Esta

organización suele recibir el nombre de archivo **disperso** o **directo**.† La condición de búsqueda debe ser una condición de igualdad sobre un solo campo, el **campo de dispersión** del archivo. Es común que el campo de dispersión sea también un campo clave del archivo, en cuyo caso se habla de la **clave de dispersión**. La dispersión se basa en establecer una función *h*, llamada **función de dispersión** o **función de aleatorización**, que se aplica al valor del campo de dispersión de un archivo y produce la *dirección* del bloque de disco en el que está almacenado el registro. La búsqueda del registro dentro del bloque puede realizarse en un *buffer* de la memoria principal. Para la mayoría de los registros, basta un solo acceso a bloque para obtener un registro.

La dispersión también sirve como estructura interna de datos en un programa siempre que se tenga acceso a un archivo temporal pequeño empleando el valor de un campo. Describiremos cómo se emplea la dispersión en archivos internos en la sección 4.8.1; luego, en la sección 4.8.2, mostraremos cómo se modifica para almacenar archivos externos en disco; en la sección 4.8.3 estudiaremos técnicas para extender la dispersión a archivos que crecen dinámicamente.

4.8.1 *Dispersión interna*

Por lo regular, la dispersión en los archivos internos se implementa con un arreglo de registros. Supongamos que el intervalo del índice del arreglo va de 0 a M – 1 (Fig. 4.8(a)); entonces, tendremos M **casillas** cuyas direcciones corresponderán a los índices del arreglo. Elegiremos una función de dispersión que transforme el valor del campo de dispersión en un entero entre 0 y M – 1. Una función de dispersión común es la función $h(K) = K \bmod M$, que devuelve el residuo, o resto, de dividir un valor entero *K* del campo de dispersión entre M; este residuo se utiliza como dirección del registro.

Los valores no enteros del campo de dispersión se pueden convertir en enteros antes de aplicar la función mod. En el caso de cadenas de caracteres, en la transformación se pueden usar los códigos numéricos asociados a los caracteres; por ejemplo, multiplicando los valores de dichos códigos. Si tenemos un campo de dispersión cuyo tipo de datos es una cadena de 20 caracteres, podemos usar el algoritmo 4.2(a) para calcular la dirección de dispersión. Suponemos que la función código devuelve el código numérico del carácter y que tenemos un valor de campo de dispersión *K* de tipo *array [1..20] of char*.

ALGORITMO 4.2 Dos algoritmos de dispersión sencillos. (a) Aplicación de la función mod a una cadena de caracteres. (b) Resolución de colisiones por direccionamiento abierto.

```
(a) temp ← 1;
    para i ← 1 a 20 hacer temp ← temp * código(K[i]);
    dirección_de_dispersión ← temp mod M;
(b) i ← dirección_de_dispersión;
    si la posición i está ocupada
       entonces comenzar i ← (i + 1) mod M;
            mientras (i ≠ dirección_de_dispersión) y la posición i está ocupada
                  hacer i ← (i + 1) mod M;
       si (i = dirección_de_dispersión) entonces todas las posiciones están
       llenas
                  si no nueva_dirección_de_dispersión ← i;
       terminar;
```

†En el sistema VAX RMS de Digital Equipment Corporation, el término *acceso directo* se refiere al acceso a un archivo relativo por posición de registro.

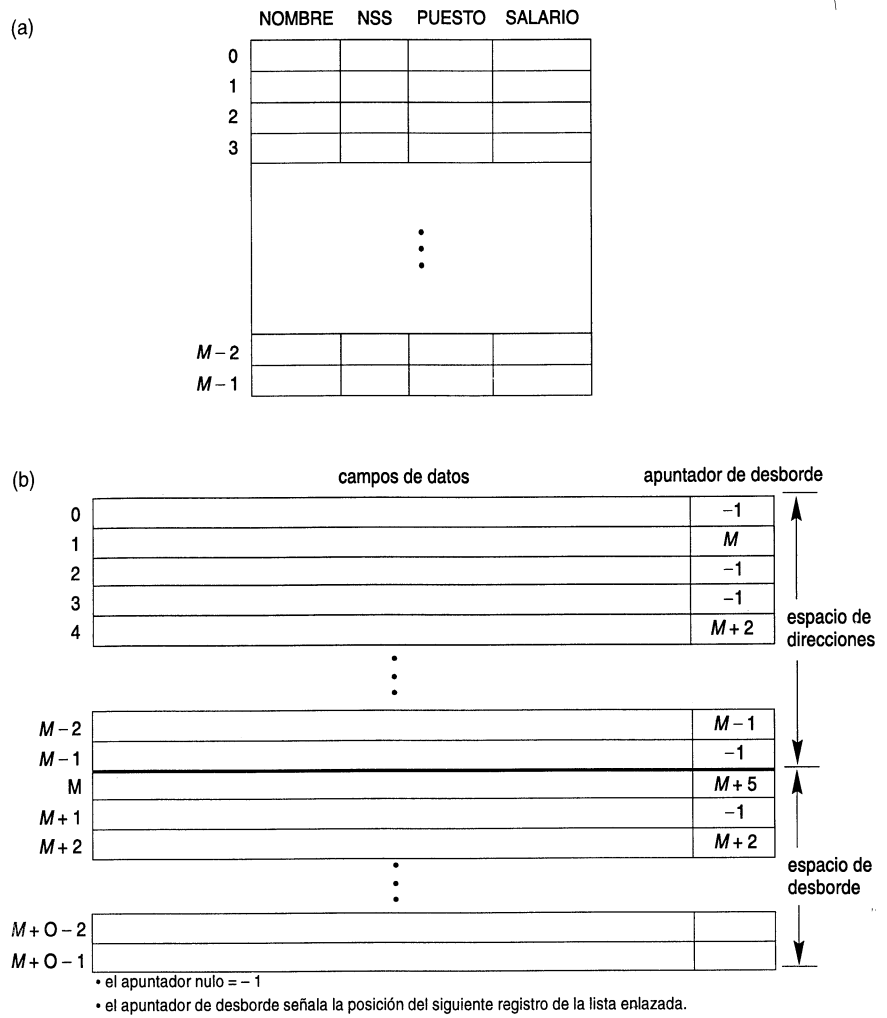


Figura 4.8 Estructuras de datos de dispersión interna. (a) Arreglo de M posiciones que se usará para la dispersión interna. (b) Resolución de colisiones por encadenamiento de registros.

Podemos usar otras funciones de dispersión. Una técnica, llamada **plegado** (*folding*), consiste en aplicar una función aritmética como la suma o una función lógica (como el "o exclusivo") a diferentes partes del valor del campo de dispersión para calcular la dirección de dispersión. Otra técnica consiste en escoger algunos dígitos del valor del campo de dispersión —por ejemplo, los dígitos tercero, quinto y octavo— para formar la dirección de dispersión. El problema con la mayoría de las funciones de dispersión es que éstas no garantizan que valores distintos produzcan direcciones de dispersión distintas, porque el **espacio del campo de dispersión** —el número de valores posibles que puede adoptar un campo de dispersión— suele ser mucho más grande que el **espacio de direcciones** —el número de direcciones disponibles para los registros—. La función de dispersión establece una transformación entre el espacio del campo de dispersión y el espacio de direcciones.

Una **colisión** se presenta cuando el valor del campo de dispersión de un registro nuevo que se desea insertar se transforma en una dirección que ya contiene otro registro. En esta situación, tendremos que insertar el registro nuevo en alguna otra posición, pues su dirección de dispersión ya está ocupada. El proceso de encontrar otra posición se denomina **resolución de colisiones**. Hay varios métodos para resolver las colisiones, entre ellos los siguientes:

- *Direccionamiento abierto:* Partiendo de la posición ocupada que especifica la dirección de dispersión, el programa examina las posiciones subsecuentes en orden hasta encontrar una posición no utilizada (vacía). El algoritmo 4.2(b) puede servir para este fin.
- *Encadenamiento:* Cuando se usa este método, se mantienen algunas áreas de desborde, por lo regular mediante la extensión del arreglo con varias posiciones de desborde. Además, se agrega un campo apuntador a cada posición de registro. Las colisiones se resuelven colocando el registro nuevo en una posición de desborde desocupada y haciendo que el apuntador de la posición ocupada, que le correspondería por su dirección de dispersión, apunte a la posición de desborde. Así, se mantiene una lista enlazada de registros de desborde para cada dirección de dispersión, como se ilustra en la figura 4.8(b).
- *Dispersión múltiple:* El programa aplica una segunda función de dispersión si la primera produce una colisión. Si resulta otra colisión, el programa usa direccionamiento abierto o aplica una tercera función de dispersión y luego utiliza direccionamiento abierto si es necesario.

Cada método de resolución de colisiones requiere sus propios algoritmos para insertar, obtener y eliminar registros. Los algoritmos del encadenamiento son los más simples; los algoritmos de eliminación del direccionamiento abierto son bastante complicados. Los textos sobre estructuras de datos analizan los algoritmos de dispersión interna con mayor detalle.

El objetivo de una buena función de dispersión es distribuir los registros uniformemente en el espacio de direcciones de modo que haya un mínimo de colisiones y a la vez se ocupen la mayor parte de las posiciones. Hay estudios de simulación y análisis que han demostrado que lo mejor suele ser mantener la tabla de dispersión ocupada entre el 70% y el 90%, para que el número de colisiones sea bajo y no se desperdicie demasiado espacio. Por tanto, si esperamos tener r registros almacenados en la tabla, deberemos escoger M posiciones para el espacio de direcciones de modo que (r/M) esté entre 0.7 y 0.9. También puede ser conveniente elegir un número primo para M, pues se ha demostrado que así las direcciones de dispersión se distribuyen mejor dentro del espacio de direcciones cuando la función utilizada para dispersar es mod. Otras funciones de dispersión tal vez requieran que M sea una potencia de 2.

4.8.2 *Dispersión externa*

La dispersión en archivos de disco se denomina **dispersión externa**. A fin de adecuarlo a las características del almacenamiento en disco, el espacio de direcciones destino se divide en **cubetas**, cada una de las cuales contiene varios registros. Cada cubeta es un bloque de disco o bien un grupo de bloques contiguos. La función de dispersión establece una transformación entre la clave y un número de cubeta relativo, en vez de asignar una dirección de bloque absoluta a la cubeta. Una tabla que se mantiene en el descriptor del archivo convierte el número de cubeta en la dirección de bloque en disco correspondiente, como se ilustra en la figura 4.9.

El problema de las colisiones es menos grave cuando se usan cubetas, porque tantos registros como quepan en una cubeta podrán dispersarse a la misma cubeta sin causar problemas. Sin embargo, deberemos prevenir el caso en que una cubeta se ocupe totalmente y un registro nuevo que se desee insertar se disperse a esa cubeta. Podemos emplear una variación del encadenamiento en la que mantengamos en cada cubeta un apuntador a una lista enlazada de registros de desborde para esa cubeta, como se muestra en la figura 4.10. Los apuntadores de la lista enlazada deberán ser **apuntadores a registros**, que incluyan tanto una dirección de bloque como una posición de registro relativa dentro del bloque.

Aunque la dispersión ofrece el acceso más rápido posible para obtener un registro arbitrario, dado el valor de su campo de dispersión, no resulta muy útil cuando se requieren otras aplicaciones del mismo archivo, a menos que se construyan caminos de acceso adicionales. Por ejemplo, si queremos leer registros en orden según los valores de su campo de dispersión, la dispersión no es muy adecuada, porque la mayor parte de las funciones de dispersión buenas no mantienen los registros en dicho orden. Algunas funciones de dispersión, las que **conservan el orden**, pueden mantener los registros ordenados según los valores del campo de dispersión. Un ejemplo sencillo es tomar los tres primeros dígitos de un campo de número de factura como la dirección de dispersión y mantener los registros ordenados por número de factura dentro de cada cubeta. Otro ejemplo sería usar una clave de dispersión entera directamente como índice de un archivo relativo, si los valores de dicha clave llenan un cierto intervalo; por ejemplo, si una compañía asigna a sus empleados los números 1, 2, 3, ..., etc.,

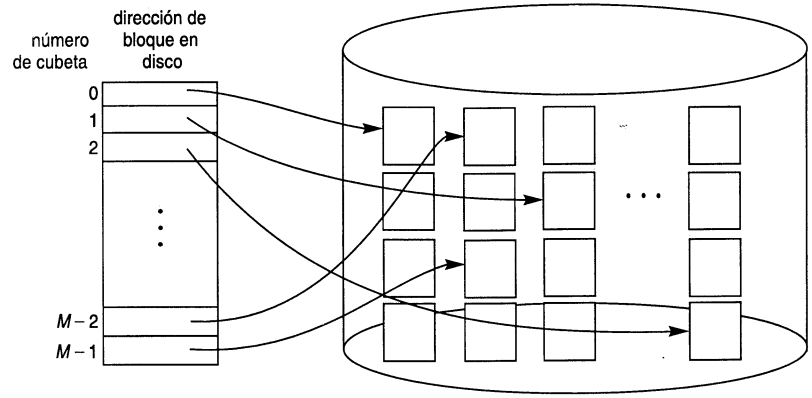


Figura 4.9 Correspondencia entre números de cubeta y bloques de disco.

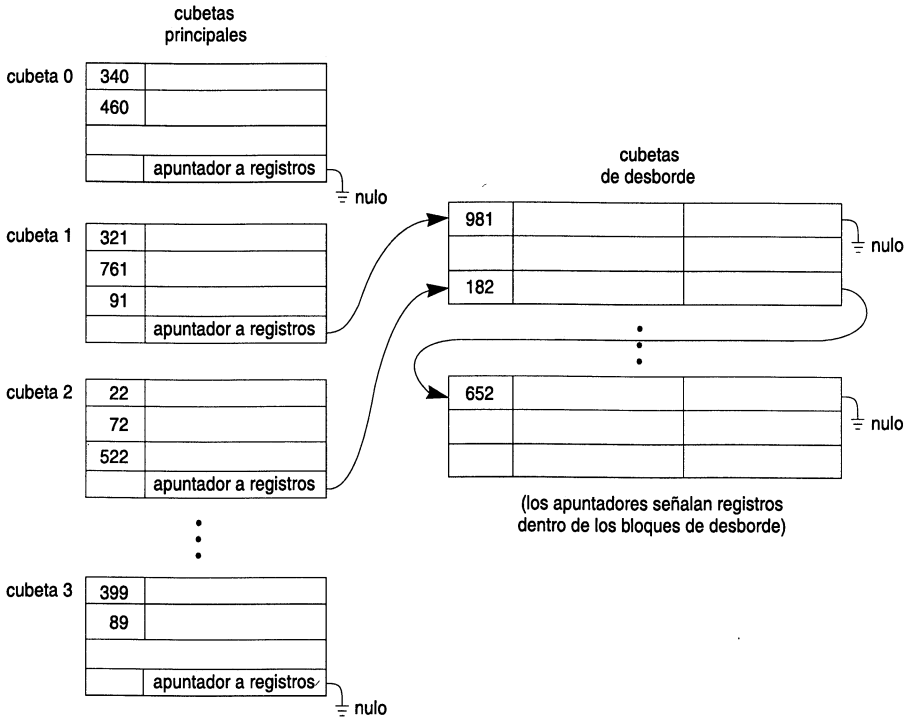


Figura 4.10 Manejo del desborde de cubetas por encadenamiento.

hasta el número total de empleados, podemos usar la función de dispersión de identidad que mantiene el orden. Desafortunadamente, esto sólo funciona si la aplicación genera las claves en orden.

Otra desventaja de la dispersión es la cantidad fija de espacio asignada al archivo. Supongamos que asignamos M cubetas al espacio de direcciones y m es el número máximo de registros que caben en una cubeta; en tal caso, cabrán a lo más m * M registros en el espacio asignado. Si resulta que el número de registros es bastante menor que m * M, tendremos mucho espacio desaprovechado. Por otro lado, si el número de registros rebasa sustancialmente la cifra m * M, habrá una gran cantidad de colisiones y la obtención de registros se hará lenta debido a las largas listas de registros de desborde. En ambos casos, quizá sea necesario modificar el número de bloques asignados y luego emplear una función de dispersión distinta para redistribuir los registros entre las cubetas. Las organizaciones de archivos más recientes basadas en la dispersión permiten que el número de cubetas varíe dinámicamente; analizaremos algunas de estas técnicas en la sección 4.8.3.

En la dispersión externa normal, la búsqueda de un registro con base en el valor de un campo distinto del de dispersión es tan costosa como en el caso de un archivo no ordenado. La eliminación de registros se puede implementar sacando el registro de su cubeta. Si ésta tiene una cadena de desborde, podemos pasar uno de los registros de desborde a la cubeta para reemplazar el registro eliminado. Si el registro por eliminar ya está en el área de desborde,

bastará con quitarlo de la lista enlazada. Adviértase que la eliminación de un registro de este tipo implica seguir la pista de las posiciones vacías del área de desborde. Esto se logra con facilidad manteniendo una lista enlazada de posiciones de desborde desocupadas.

La modificación del valor de un campo de un registro depende de dos factores: la condición de búsqueda que se usa para localizar el registro y el campo que se va a modificar. Si la condición de búsqueda es una comparación de igualdad con el campo de dispersión, podremos localizar el registro con eficiencia mediante la función de dispersión; en caso contrario, será preciso efectuar una búsqueda lineal. Los campos que no se usan para la dispersión pueden modificarse actualizando el registro y reescribiéndolo en la misma cubeta. Modificar el campo de dispersión implica que el registro pueda pasar a otra cubeta, para lo cual habría que eliminar el registro antiguo e insertar el registro modificado.

4.8.3 *Técnicas de dispersión que permiten la expansión dinámica de los archivos*★

Una desventaja importante del esquema de dispersión *estática* que acabamos de ver es que el espacio de direcciones de dispersión es fijo, lo que dificulta la expansión o contracción dinámicas del archivo. Los esquemas que describiremos en esta sección intentan remediar este problema. Los primeros dos esquemas, la dispersión dinámica y la dispersión extensible, almacenan una estructura de acceso además del archivo, lo que los hace un tanto similares a la indización (Cap. 5). La diferencia principal es que la estructura de acceso se basa en los valores que resultan de aplicar la función de dispersión al campo de búsqueda. En la indización, la estructura de acceso se basa en el valor del campo de búsqueda mismo. La tercera técnica, la dispersión lineal, no requiere ninguna estructura de acceso adicional.

Estos esquemas de dispersión aprovechan el hecho de que la mayor parte de las funciones de dispersión dan como resultado un entero no negativo, que puede representarse como número binario. La estructura de acceso se basa en la **representación binaria** del resultado de la función de dispersión, que es una cadena de **bits** a la cual llamamos **valor de dispersión** de un registro. Los registros se distribuyen entre las cubetas según los valores de los *bits más significativos* de sus valores de dispersión.

Dispersión dinámica. En la dispersión dinámica, el número de cubetas no es fijo (como en la dispersión normal), sino que aumenta o disminuye según las necesidades. El archivo puede comenzar con una sola cubeta; una vez que ésta se llena y se inserta un nuevo registro, la cubeta se **desborda** y se divide en dos. Los registros se distribuyen entre ambas con base en el valor del primer bit (el del extremo izquierdo) de sus valores de dispersión. Los registros cuyos valores de dispersión comiencen con el bit 0 se colocarán en una cubeta, y los que comiencen con 1 se almacenarán en la otra. En este momento se construye una estructura de árbol binario llamada **directorio** (o **índice**). El directorio tiene dos tipos de nodos:

- Los **nodos internos** guían la búsqueda; cada uno tiene un apuntador izquierdo que corresponde a un bit 0 y un apuntador derecho que corresponde a un bit 1.
- Los **nodos hoja** contienen un apuntador a una cubeta, esto es, una dirección de cubeta. La figura 4.11 ilustra un directorio y las cubetas de un archivo de datos.

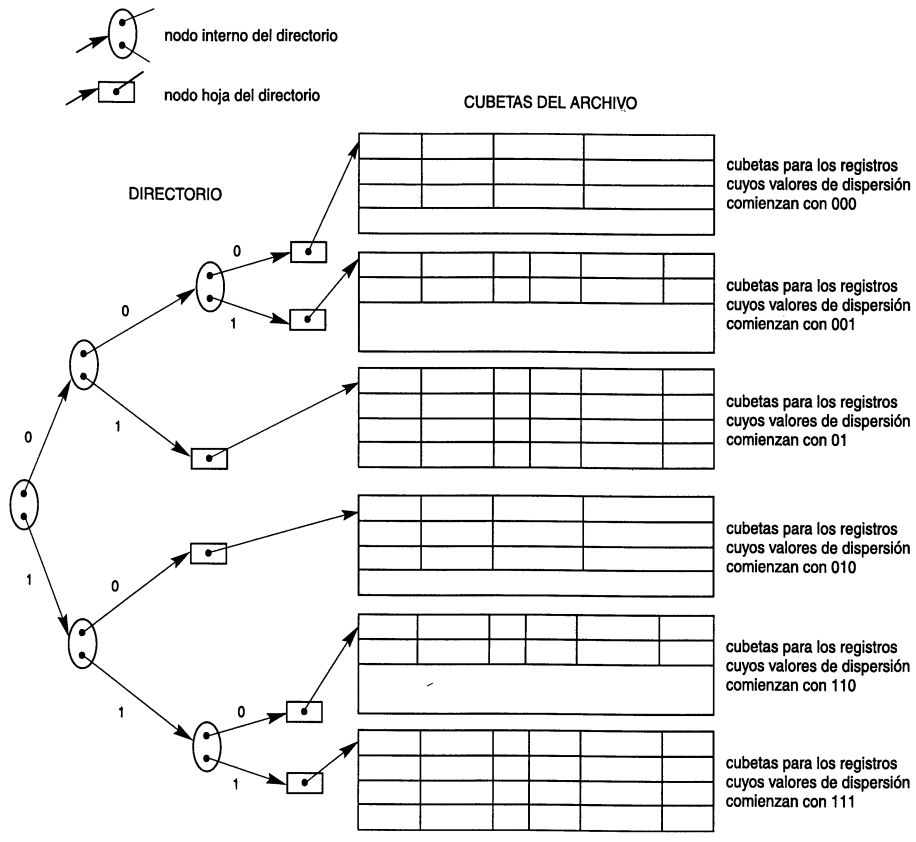


Figura 4.11 Estructura del esquema de dispersión dinámica.

ALGORITMO 4.3 Procedimiento de búsqueda para la dispersión dinámica.

```
h ← valor de dispersión del registro;
t ← nodo raíz del directorio;
i ← 1;
mientras t sea un nodo interno del directorio hacer
    comenzar
    si el i-ésimo bit de h es un 0
        entonces t ← hijo izquierdo de t
        si no t ← hijo derecho de t;
    i ← i + 1
    terminar;
buscar en la cubeta cuya dirección está en el nodo t;
```

La búsqueda de un registro se efectúa como indica el algoritmo 4.3. El directorio se puede almacenar en la memoria principal a menos que crezca demasiado. Si el directorio no cabe en un bloque, se distribuirá en dos o más niveles. Obsérvese que las entradas del directorio son bastante compactas. Cada nodo interno contiene un bit de etiqueta para especificar el tipo de nodo, más los apuntadores izquierdo y derecho. También es posible que haga falta un apuntador al padre. Cada nodo hoja contiene una dirección de cubeta. Podemos emplear representaciones especiales de árboles binarios para reducir el espacio requerido para los apuntadores izquierdo, derecho y al padre de los nodos internos. En general, si se almacena en disco un directorio de x niveles, se requerirán $x + 1$ accesos de bloque para obtener el contenido de una cubeta.

Si una cubeta se desborda, se divide en dos y los registros se distribuyen con base en el siguiente bit más significativo de sus valores de dispersión. Por ejemplo, si se inserta un nuevo registro en la cubeta de los registros cuyos valores de dispersión comienzan con 10 (la cuarta cubeta en la figura 4.11) y esto provoca un desborde, todos los registros cuyos valores de dispersión comiencen con 100 se colocarán en la primera de las cubetas divididas, y la segunda cubeta contendrá los registros cuyos valores de dispersión comiencen con 101. El directorio se expandirá con un nuevo nodo interno a fin de reflejar la división; este nodo apuntará a dos nodos hoja que apuntarán a las dos cubetas. Así, los niveles del árbol binario se pueden expandir dinámicamente; sin embargo, el número de niveles no puede exceder el número de bits del valor de dispersión.

Si la función de dispersión distribuye los registros de manera uniforme, el árbol del directorio estará equilibrado. Es posible combinar dos cubetas si una de ellas se vacía o si el total de registros de dos cubetas vecinas puede caber en una sola. En este caso, el directorio perderá un nodo interno y los dos nodos hoja se combinarán para formar un solo nodo hoja que apunte a la nueva cubeta. Así, los niveles del árbol binario se pueden contraer dinámicamente.

Dispersión extensible. En la dispersión extensible se mantiene un tipo distinto de **directorio**, un arreglo de 2^d direcciones de cubeta, donde d es la **profundidad global** del directorio. El valor entero que corresponde a los primeros d bits (los más significativos) de un valor de dispersión sirve como índice del arreglo para determinar una entrada del directorio, y la dirección contenida en esa entrada determinará la cubeta en la que se almacenarán los registros correspondientes. Pero no tiene que haber una cubeta distinta para cada una de las 2^d posiciones del directorio. Varias posiciones del directorio que tengan los mismos primeros d' bits en sus valores de dispersión pueden contener la misma dirección de cubeta si todos los registros que se dispersan a esas posiciones caben en una sola cubeta. Una **profundidad local** d' —almacenada con cada cubeta— especifica el número de bits en el que se basa el contenido de la cubeta. La figura 4.12 muestra un directorio de profundidad global $d = 3$.

El valor de d se puede aumentar o reducir en uno a la vez, con lo cual se duplicará o reducirá a la mitad el número de entradas del arreglo del directorio. Será necesario duplicarlas si se desborda una cubeta cuya profundidad local d' es igual a la profundidad global d. Se podrán reducir a la mitad si $d > d'$ para todas las cubetas después de efectuarse algunas eliminaciones. La obtención de un registro requerirá en general dos accesos a bloque: uno al directorio y otro a la cubeta.

Para ilustrar la división de una cubeta, supongamos que la inserción de un registro nuevo provoca el desborde de la cubeta cuyos valores de dispersión comienzan con 01 (la tercera cubeta en la figura 4.12). Los registros se distribuirán entre dos cubetas: la primera contendrá todos los registros cuyos valores de dispersión comiencen con 010, y la segunda

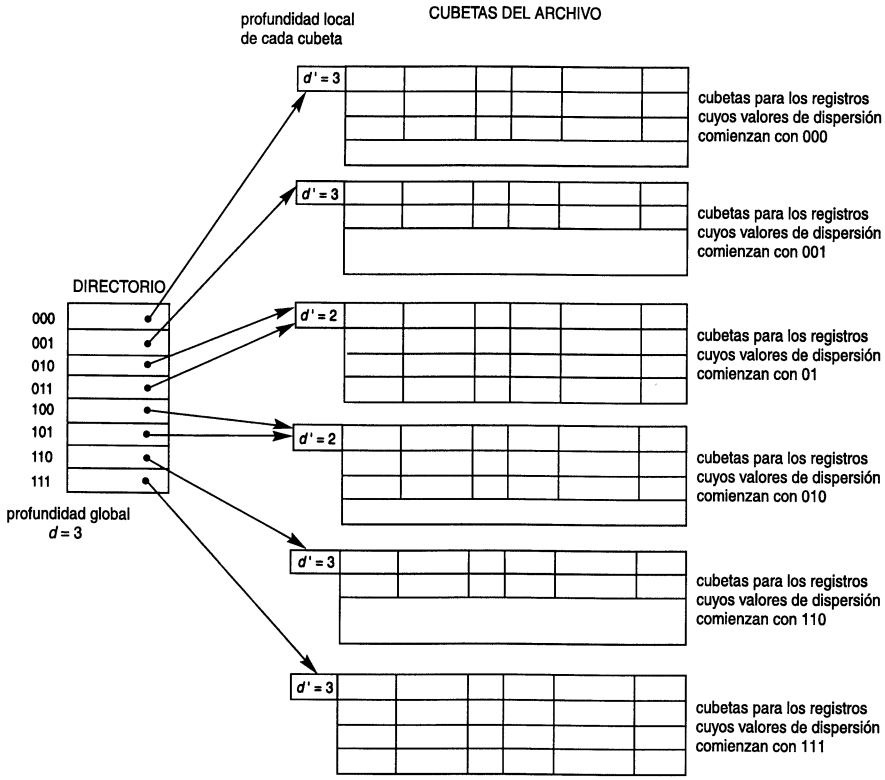


Figura 4.12 Estructura del esquema de dispersión extensible.

contendrá todos aquellos cuyos valores de dispersión comiencen con 011. Ahora, las dos posiciones de directorio correspondientes a 010 y 011, que antes de la división apuntaban a la misma cubeta, apuntarán ahora a dos cubetas distintas. La profundidad local d' de las dos nuevas cubetas será 3: uno más que la profundidad local de la antigua cubeta.

Si se desborda y divide una cubeta con una profundidad local d' igual a la profundidad global d del directorio, será preciso duplicar el tamaño del directorio a fin de poder usar un bit más para distinguir las dos nuevas cubetas. Por ejemplo, si se desborda la cubeta de la figura 4.12 que contiene los registros cuyos valores de dispersión comienzan con 111, las dos cubetas nuevas requerirán un directorio con profundidad global $d = 4$, porque ahora corresponden a los valores de dispersión que comienzan con 1110 y 1111, de modo que su profundidad local es 4. Por tanto, se duplica el tamaño del directorio, y todas las demás posiciones del directorio original se dividen también en dos, aunque ambas tendrán el mismo apuntador que tenía la posición original. El tamaño máximo del directorio es 2^k, donde k es el número de bits del valor de dispersión.

Dispersión lineal. La dispersión lineal se basa en la idea de permitir que el archivo de dispersión aumente y reduzca dinámicamente el número de cubetas *sin* necesidad de un directorio.

Supongamos que el archivo comienza con M cubetas numeradas 0, 1, ..., M – 1 y utiliza la función de dispersión $h(K) = K \bmod M$; esta función se denomina función de dispersión inicial h_0. El desborde ocasionado por colisiones se resuelve manteniendo cadenas de desborde individuales para cada cubeta, con la diferencia de que cuando una colisión origina un registro de desborde en *cualquier* cubeta, la *primera* cubeta del archivo —la cubeta 0— se divide en dos: la cubeta 0 original y una nueva cubeta M al final del archivo. Los registros que originalmente estaban en la cubeta 0 se distribuyen entre las dos cubetas con base en una función de dispersión distinta $h_1(K) = K \bmod 2M$. Una propiedad clave de las dos funciones h_0 y h_1 es que todos los registros que se dispersaban a la cubeta 0 con base en h_0 se dispersarán a la cubeta 0 o a la M con base en h_1; esto es necesario para que funcione la dispersión lineal.

Al ocurrir más colisiones que originen registros de desborde, se dividirán cubetas adicionales en el orden *lineal* 1, 2, 3, ... Si hay suficientes desbordes, se dividirán todas las cubetas del archivo, y así todos los registros de desborde se redistribuirán en cubetas normales con la función h_1 mediante una *división postergada* de sus cubetas. No se requiere directorio alguno, sólo un valor n para determinar cuáles cubetas se han dividido ya. Si queremos recuperar un registro cuyo valor de la clave de dispersión sea K, primero aplicamos la función h_0 a K; si $h_0(K) < n$, aplicaremos la función h_1 a K porque la cubeta ya está dividida. En un principio, $n = 0$, lo que indica que la función h_0 se aplica a todas las cubetas; n crece linealmente conforme se van dividiendo las cubetas.

Cuando $n = M$, se habrán dividido todas las cubetas originales y la función de dispersión h_1 se aplicará a todos los registros del archivo. En este punto, n se pone en 0 otra vez, y cualesquier colisiones nuevas que provoquen desborde harán que se utilice una nueva función de dispersión $h_2(K) = K \bmod 4M$. En general, se utilizará una secuencia de funciones de dispersión $h_j(K) = K \bmod (2^jM)$, donde $j = 0, 1, 2, ...$; se requerirá una nueva función de dispersión h_{j+1} cuando se hayan dividido todas las cubetas 0, 1, ..., $(2^jM) - 1$ y n se haya puesto otra vez en 0. La búsqueda de un registro cuyo valor de clave de dispersión sea K se efectúa con el algoritmo 4.4.

Las cubetas que se han dividido pueden recombinarse si la carga del archivo cae por debajo de cierto umbral. En general, se puede definir el **factor de carga** de un archivo, l, como $l = r/(fbl * N)$, donde r es el número actual de registros del archivo, fbl es el número máximo de registros que pueden caber en una cubeta y N es el número actual de cubetas del archivo. Los bloques se combinan linealmente, y N se reduce de acuerdo con ello. La carga del archivo puede servir para iniciar tanto divisiones como recombinaciones; así, la carga del archivo se mantiene dentro de un intervalo deseado. Las divisiones pueden iniciarse cuando la carga exceda un cierto umbral —digamos 0.9— y las combinaciones cuando la carga caiga por debajo de otro umbral, digamos 0.7.

ALGORITMO 4.4 Procedimiento de búsqueda para la dispersión lineal.

```
si n = 0
  entonces m ← h_j(K) (* m es el valor de dispersión del registro con clave de
dispersión K *)
  si no comenzar
      m ← h_j(K);
      si m<n entonces m ← h_{j+1}(K)
      terminar;
buscar en la cubeta cuyo valor de dispersión es m (y en su desborde, si lo hay);
```

4.9 Otras organizaciones primarias de archivos*

4.9.1 *Archivos de registros mixtos*

En las organizaciones de archivos que hemos visto hasta ahora se supone que todos los registros de un cierto archivo son del mismo tipo. Los registros podrían ser EMPLEADOs, PROYECTOs, ESTUDIANTEs o DEPARTAMENTOs, pero cada archivo contiene sólo registros de un tipo. En la mayoría de las aplicaciones de bases de datos encontramos situaciones en las que muchos tipos de entidades están interrelacionados de diversas maneras, como vimos en el capítulo 3. Los vínculos entre los registros de varios archivos se pueden representar mediante **campos conectores**. Por ejemplo, un registro ESTUDIANTE puede tener un campo conector DEPTOCARRERA cuyo valor proporcione el nombre del DEPARTAMENTO en el que el estudiante se va a graduar. Este campo DEPTOCARRERA *hace referencia* a una entidad DEPARTAMENTO, la cual deberá contar con un registro propio en el archivo DEPARTAMENTO. Si queremos obtener valores de campos de dos registros vinculados, tendremos que recuperar primero uno de los registros. A continuación, podremos usar el valor de su campo conector para obtener el registro vinculado del otro archivo. Por tanto, los vínculos se implementan por medio de **referencias lógicas de campos** entre los registros de archivos distintos.

Las organizaciones de archivos en los SGBD jerárquicos y de red implementan los vínculos entre registros como **vínculos físicos,** los cuales se constituyen por la contigüidad física de los registros vinculados o mediante apuntadores físicos. Estas organizaciones suelen asignar un **área** del disco para almacenar registros de más de un tipo a fin de que registros de diferentes tipos puedan estar relacionados físicamente. Si se espera utilizar con mucha frecuencia un cierto vínculo, la implementación física de éste puede aumentar la eficiencia del sistema en cuanto a la obtención de registros relacionados. Por ejemplo, si va a ser muy común la consulta para obtener un registro de DEPARTAMENTO y todos los registros de los ESTUDIANTEs que se van a graduar en ese departamento, sería conveniente colocar cada registro DEPARTAMENTO y su grupo de registros ESTUDIANTE contiguamente en el disco, dentro de un archivo mixto.

Para distinguir los registros de un archivo mixto, cada registro tiene, además de los valores de sus campos, un campo de **tipo de registro**, el cual suele ser el primero del registro. El software del sistema utiliza este campo para conocer el tipo del registro que va a procesar. Con la ayuda de la información del catálogo, el SGBD puede determinar los campos de ese tipo de registro y sus tamaños, y así interpretar los datos del mismo.

4.9.2 *Árboles B y otras estructuras de datos*

Podemos usar otras estructuras de datos en las organizaciones primarias de los archivos. Por ejemplo, si tanto el tamaño de los registros como su número son pequeños, algunos SGBD ofrecen la opción de utilizar una estructura de datos de árbol B como organización primaria del archivo. Describiremos los árboles B en la sección 5.3.1, cuando hablemos del empleo de esa estructura de datos para la indización. En general, cualquier estructura de datos que se pueda adaptar a las características de los dispositivos de disco se podrá utilizar como organización primaria de archivo para ubicar los registros en el disco.

4.10 Resumen

Comenzamos este capítulo con un análisis de las características de los dispositivos de almacenamiento secundario. Nos concentramos en los discos magnéticos porque son los de uso más difundido para almacenar archivos de base de datos en línea. Los datos se almacenan en bloques en el disco; el acceso a un bloque de disco es costoso debido al tiempo de búsqueda, al retardo rotacional y al tiempo de transferencia de bloque. Se puede utilizar doble almacenamiento intermedio cuando se quiera tener acceso a bloques consecutivos, a fin de reducir el tiempo medio de acceso a un bloque. En el apéndice B se estudian otros parámetros de disco.

Vimos diferentes maneras de almacenar registros en archivos de disco. Los registros del archivo se agrupan en bloques y pueden tener longitud fija o variable, ser extendidos o no, y ser del mismo tipo o mixtos. Hablamos del descriptor de archivo, que describe los formatos de los registros y contiene las direcciones en disco de los bloques del archivo. El software del sistema utiliza la información del descriptor para tener acceso a los registros del archivo.

A continuación presentamos un conjunto de órdenes representativas para tener acceso a registros individuales de los archivos y analizamos el concepto de registro actual de un archivo. Vimos cómo transformar condiciones complejas para la búsqueda de registros en condiciones de búsqueda simples que sirven para localizar registros en el archivo.

Luego se estudiaron tres organizaciones primarias de los archivos: no ordenada, ordenada y dispersa. Los archivos no ordenados requieren una búsqueda lineal para localizar registros, pero la inserción de éstos es muy sencilla. Vimos el problema de la eliminación y el empleo de marcadores de eliminación.

Los archivos ordenados reducen el tiempo requerido para leer registros en orden según el campo de ordenación. El tiempo requerido para buscar un registro arbitrario, dado el valor de su campo clave de ordenamiento, también se reduce si se utiliza una búsqueda binaria. Sin embargo la necesidad de mantener los registros en orden hace muy costosa la inserción; por esta razón, se examinó la técnica de usar un archivo de desborde no ordenado para reducir el costo de la inserción. Los registros de desborde se fusionan con el archivo maestro periódicamente durante la reorganización del archivo.

La dispersión ofrece acceso muy rápido a un registro arbitrario del archivo, dado el valor de su campo de dispersión. El método más adecuado para la dispersión externa es la técnica de cubetas, en la que uno o más bloques contiguos corresponden a cada cubeta. Las colisiones que causan desborde de las cubetas se resuelven con el encadenamiento. El acceso según un campo que no sea el de dispersión es lento, y lo mismo sucede con el acceso secuencial a los registros basado en cualquier campo. Más adelante estudiamos técnicas de dispersión que permiten al archivo expandirse y contraerse dinámicamente, entre ellas la dispersión dinámica, la extensible y la lineal.

Por último, mencionamos brevemente otras posibilidades de organización primaria de los archivos, como los árboles B, y los archivos de registros mixtos, que implementan los vínculos de los registros de diferentes tipos físicamente, como parte de la estructura de almacenamiento.

Preguntas de repaso

4.1. ¿Qué diferencia hay entre almacenamiento primario y secundario?

4.2. ¿Por qué se usan discos y no cintas para almacenar archivos de bases de datos en línea?

4.3. Defina los siguientes términos: *disco, paquete de disco, pista, bloque, cilindro, sector, separación entre registros, cabeza de lectura/escritura.*

4.4. Explique el proceso de iniciación de un disco.

4.5. Analice los mecanismos empleados para leer datos de un disco o escribirlos en él.

4.6. ¿Cuáles son los componentes de una dirección de bloque de disco?

4.7. ¿Por qué resulta costoso tener acceso a un bloque de disco? Analice los componentes de tiempo que intervienen en el acceso a un bloque de disco.

4.8. ¿De qué manera el doble almacenamiento intermedio mejora el tiempo de acceso a un bloque?

4.9. ¿Qué razones hay para tener registros de longitud variable? ¿Qué tipos de caracteres separadores se requieren en cada caso?

4.10. Analice las diferentes técnicas para asignar bloques de un archivo en el disco.

4.11. ¿Qué diferencia hay entre una organización de archivo y un método de acceso?

4.12. ¿Qué diferencia hay entre una condición de selección y una condición de búsqueda?

4.13. ¿Cuáles son las operaciones comunes de registro por registro para tener acceso a un archivo? ¿Cuáles de ellas dependen del registro actual del archivo?

4.14. Analice las diferentes técnicas para eliminar registros.

4.15. Comente las ventajas y desventajas de utilizar (a) un archivo no ordenado, (b) un archivo ordenado y (c) un archivo de dispersión normal (estática) con cubetas y encadenamiento. ¿Cuáles operaciones se pueden ejecutar eficientemente con cada una de esas organizaciones, y cuáles resultan costosas?

4.16. Analice las diferentes técnicas para permitir que un archivo de dispersión se expanda o contraiga dinámicamente. ¿Qué ventajas y desventajas tiene cada una?

4.17. ¿Para qué se usan los archivos mixtos? ¿Qué otros tipos de organización primaria de archivos hay?

Ejercicios

4.18. Considere un disco con las siguientes características (no se trata de parámetros de ninguna unidad de disco en particular): tamaño de bloque $B = 512$ bytes; tamaño de la separación entre bloques $G = 128$ bytes; número de bloques por pista = 20; número de pistas por superficie = 400. Un paquete de discos consta de 15 discos de dos lados cada uno.

a. ¿Cuál es la capacidad total de una pista, y cuál su capacidad útil (excluyendo las separaciones entre registros)?

b. ¿Cuántos cilindros hay?

c. ¿Cuál es la capacidad total y la capacidad útil de un cilindro?

d. ¿Cuál es la capacidad total y la capacidad útil de un paquete de discos?

e. Suponga que la unidad de disco gira el paquete a una velocidad de 2400 rpm (revoluciones por minuto); ¿cuál es la velocidad de transferencia en bytes/milisegundo y el tiempo de transferencia de bloques (*ttb*) en milisegundos? ¿Cuál es el retardo rotacional (*rr*) medio en milisegundos? ¿Cuál es la velocidad de transferencia masiva (véase el apéndice B)?

f. Suponga que el tiempo de búsqueda medio es de 30 milisegundos. ¿Cuántos milisegundos tarda (en promedio) la localización y transferencia de un solo bloque, dada su dirección de bloque?

g. Calcule el tiempo medio que tardaría la transferencia de 20 bloques aleatorios, y compárelo con el tiempo que tardaría la transferencia de 20 bloques consecutivos empleando doble almacenamiento intermedio para ahorrar tiempo de búsqueda y retardo rotacional.

4.19. Un archivo tiene $r = 20\ 000$ registros ESTUDIANTE de *longitud fija*. Cada registro tiene los siguientes campos: NOMBRE (30 bytes), NSS (9 bytes), DIRECCIÓN (40 bytes), TELÉFONO (9 bytes), FECHANACIMIENTO (8 bytes), SEXO (1 byte), CÓDIGODEPTOCARRERA (4 bytes), CÓDIGODEPTOESPECIALIDAD (4 bytes), CÓDIGOGRADO (4 bytes, entero) y PROGRAMAGRADO (3 bytes). Se utiliza un byte adicional como marcador de eliminación. El archivo se almacena en el disco cuyos parámetros se dan en el ejercicio 4.18.

a. Calcule el tamaño de un registro R en bytes.

b. Calcule el factor de bloques (*fbl*) y el número de bloques del archivo (b), suponiendo una organización no extendida.

c. Calcule el tiempo medio que tarda la búsqueda de un registro si se utiliza una búsqueda lineal y si (i) los bloques del archivo se almacenan contiguamente y se utiliza doble almacenamiento intermedio; (ii) los bloques del archivo no se almacenan contiguamente.

d. Suponga que el archivo está ordenado por NSS; calcule el tiempo que tarda la búsqueda de un registro dado su valor de NSS, si se utiliza una búsqueda binaria.

4.20. Suponga que sólo el 80% de los registros ESTUDIANTE del ejercicio 4.19 tienen un valor en el campo TELÉFONO, el 85% en CÓDIGODEPTOCARRERA, el 15% en CÓDIGODEPTOESPECIALIDAD y el 90% en PROGRAMAGRADO; suponga además que se usa un archivo de registros de longitud variable. Cada registro tiene un *tipo de campo* de un byte por cada campo incluido en el registro, más el marcador de eliminación de un byte y un marcador de fin de registro de un byte. Suponga que utilizamos una organización de registros *extendidos*, en la que cada bloque tiene un apuntador de cinco bytes al siguiente bloque (este espacio no se utiliza para almacenar registros).

a. Calcule la longitud media (R) de los registros en bytes.

b. Calcule el número de bloques requeridos para el archivo.

4.21. Suponga que una unidad de disco tiene los siguientes parámetros: tiempo de búsqueda $s = 20$ ms; retardo rotacional $rr = 10$ ms; tiempo de transferencia de bloques $ttb =$

1 ms; tamaño de bloque $tb = 2400$ bytes; tamaño de la separación entre bloques $G = 600$ bytes. Un archivo EMPLEADO tiene los siguientes campos: NSS, 9 bytes; APELLIDO, 20 bytes; NOMBRE, 20 bytes; INICIAL, 1 byte; FECHANAC, 10 bytes; DIRECCIÓN, 35 bytes; TELÉFONO, 12 bytes; NSSSUPERVISOR, 9 bytes; DEPARTAMENTO, 4 bytes; CÓDIGOPUESTO, 4 bytes; *marcador de eliminación*, 1 byte. El archivo EMPLEADO tiene $r = 30\ 000$ registros de longitud fija y bloques no extendidos. Escriba fórmulas apropiadas y calcule los siguientes valores para este archivo EMPLEADO:

a. El tamaño de registro, R (incluido el marcador de eliminación), el factor de bloques, *fbl*, y el número de bloques de disco, b.

b. Calcule el espacio desperdiciado en cada bloque de disco debido a la organización no extendida.

c. Calcule la velocidad de transferencia, *vt*, y la velocidad de transferencia masiva, *vtm* para esta unidad de disco. (Véase en el apéndice B las definiciones de *vt* y de *vtm*.)

d. Calcule el *número medio de accesos a bloque* necesarios para buscar un registro arbitrario en el archivo, mediante búsqueda lineal.

e. Calcule el *tiempo* medio requerido (en ms) para buscar un registro arbitrario en el archivo, empleando búsqueda lineal, si los bloques del archivo se almacenan en bloques consecutivos del disco y se utiliza doble almacenamiento intermedio.

f. Calcule el *tiempo* medio requerido (en ms) para buscar un registro arbitrario en el archivo, mediante búsqueda lineal, si los bloques del archivo *no* están almacenados en bloques consecutivos del disco.

g. Suponga que los registros están ordenados según un campo clave. Calcule el *número medio de accesos a bloque* y el *tiempo medio* necesario para buscar un registro arbitrario en el archivo, empleando búsqueda binaria.

4.22. Un archivo COMPONENTES con NúmComp como clave de dispersión contiene registros con los siguientes valores de NúmComp: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981, 9208. El archivo utiliza ocho cubetas, numeradas del 0 al 7. Cada cubeta es un bloque de disco y contiene dos registros. Cargue estos registros en el archivo en el orden dado, empleando la función de dispersión $h(K) = K \bmod 8$. Calcule el número medio de accesos a bloque para una obtención aleatoria con base en NúmComp.

4.23. Cargue los registros del ejercicio 4.22 en archivos de dispersión expansible basados en (i) dispersión dinámica y (ii) dispersión extensible. Muestre la estructura del directorio en cada paso. En el caso de la dispersión extensible, muestre las profundidades locales y globales en cada etapa. Utilice la función de dispersión $h(K) = K \bmod 32$.

4.24. Cargue los registros del ejercicio 4.22 en un archivo de dispersión expansible, empleando dispersión lineal. Comience con un solo bloque de disco, empleando la función de dispersión $h_0(K) = K \bmod 2^0$, y muestre cómo crece el archivo y cómo cambian las funciones de dispersión conforme se insertan los registros. Suponga que los bloques se dividen siempre que hay desborde, y muestre el valor de n en cada etapa.

4.25. Compare las órdenes de archivo mencionadas en la sección 4.5 con las disponibles en un método de acceso a archivos que conozca.

4.26. Suponga que tiene un archivo no ordenado de registros de longitud fija que utiliza una organización de registros no extendidos. Describa a grandes rasgos algoritmos para la inserción, la eliminación y la modificación de un registro de ese archivo. Exprese todas las suposiciones que haga.

4.27. Suponga que tiene un archivo ordenado de registros de longitud fija y un archivo de desborde no ordenado para manejar la inserción. Ambos archivos utilizan registros no extendidos. Describa a grandes rasgos algoritmos para la inserción, la eliminación y la modificación de un registro de ese archivo, y para reorganizarlo. Exprese todas las suposiciones que haga.

4.28. ¿Se le ocurren otras técnicas además del archivo de desborde no ordenado que pudieran servir para hacer más eficiente la inserción en un archivo ordenado?

4.29. Suponga que tiene un archivo de dispersión de registros de longitud fija, y que el desbordamiento se maneja por encadenamiento. Describa a grandes rasgos algoritmos para la inserción, la eliminación y la modificación de un registro de ese archivo. Exprese todas las suposiciones que haga.

4.30. ¿Se le ocurren otras técnicas aparte del encadenamiento para manejar el desborde de las cubetas en la dispersión externa?

4.31. Escriba un segmento de programa que sirva para tener acceso a campos individuales de los registros en cada una de las circunstancias que se describen. En cada caso, exprese las suposiciones que haga en cuanto a apuntadores, caracteres separadores, etc. Determine los tipos de información que debe incluir el descriptor del archivo para que el programa sea general en cada caso.

a. Registros de longitud fija con bloques no extendidos.

b. Registros de longitud fija con bloques extendidos.

c. Registros de longitud variable con campos de longitud variable y bloques extendidos.

d. Registros de longitud variable con grupos repetitivos y bloques extendidos.

e. Registros de longitud variable con campos opcionales y bloques extendidos.

f. Registros de longitud variable que contemplen los tres casos de las partes c, d y e.

Bibliografía selecta

Wiederhold (1983) contiene un análisis detallado de los dispositivos de almacenamiento secundario y de las organizaciones de archivo. Los discos ópticos se describen en Berg y Roth (1989) y se analizan en Ford y Christodoulakis (1991). Otros libros de texto, listados en la bibliografía al final de los capítulos 1 y 2, incluyen tratamientos del material aquí presentado. La mayoría de los textos sobre estructuras de datos, como Knuth (1973), tratan la dispersión estática con mayor detalle; Knuth tiene un análisis completo de las funciones de dispersión y de las técnicas de resolución de colisiones, así como de su rendimiento comparativo. Knuth ofrece además un detallado estudio de las técnicas para ordenar archivos externos. Salzberg *et al.* (1991) describe un algoritmo de ordenación externa distribuida.

Morris (1968) es uno de los primeros artículos sobre dispersión. La dispersión dinámica se debe a Larson (1978), y la dispersión extensible se describe en Fagin *et al.* (1979). La dispersión lineal se describe en Litwin (1980). Se han propuesto muchas variaciones de las dispersiones dinámica, extensible y lineal; por ejemplo, véase Cesarini y Soda (1991), Du y Tong (1991), y Hachem y Berra (1992).

Han aparecido varios libros de texto cuyo tema principal es las organizaciones de archivos y los métodos de acceso: Smith y Barnes (1987), Salzberg (1988), Miller (1987) y Livadas (1989).

CAPÍTULO 5

Estructuras de índices para archivos

En este capítulo describiremos las **estructuras de acceso** llamadas **índices**, que sirven para agilizar la obtención de registros en respuesta a ciertas condiciones de búsqueda. Hay unos tipos de índices, los denominados **caminos secundarios de acceso**, que no afectan la colocación física de los registros en el disco; más bien, ofrecen caminos alternativos de búsqueda para localizar eficientemente los registros con base en los **campos de indización**. Otros tipos de índices sólo se pueden construir si el archivo tiene una cierta organización primaria. En general, con cualquiera de las estructuras de datos analizadas en el capítulo 4 podemos construir un camino secundario de acceso, pero los tipos de índices más utilizados se basan en archivos ordenados (índices de un solo nivel) y en estructuras de datos de árbol (índices de múltiples niveles, árboles B^+). También es posible construir índices con base en la dispersión o en otras estructuras de datos.

Describiremos varios tipos de índices de un solo nivel —primarios, secundarios y de agrupamiento— en la sección 5.1. En la 5.2 mostraremos cómo los propios índices de un solo nivel se pueden considerar como archivos ordenados, y presentaremos el concepto de índices de múltiples niveles. En la sección 5.3 describiremos los árboles B y B^+, tan utilizados para implementar índices de múltiples niveles con cambios dinámicos. En la sección 5.4 estudiaremos cómo aprovechar otras estructuras de datos, como la dispersión, para construir índices. También explicaremos la diferencia entre los índices lógicos y los físicos.

5.1 Tipos de índices ordenados de un solo nivel

Las estructuras de acceso de índice ordenado se basan en una idea similar a la de los índices analíticos en que vemos el contenido de cualquier libro de texto: listas en orden alfabético de los términos importantes, que aparecen al final del libro. Junto a cada término viene una lista de las páginas donde aparece dicho término. Y así podemos examinar el índice para

encontrar una lista de *direcciones* —números de página en este caso— y usar estas direcciones para localizar el término en el texto *buscando* en las páginas especificadas. La alternativa, si no se cuenta con alguna otra guía, es leer todo el libro palabra por palabra hasta encontrar el término que nos interesa; esto sería equivalente a efectuar una búsqueda lineal en un archivo. Desde luego, la mayoría de los libros ofrecen información adicional, como los títulos de capítulos y secciones, que pueden ayudarnos a localizar un término sin tener que buscar en todo el libro. Sin embargo, el índice es la única señalización exacta de dónde aparece cada término en el libro.

Las estructuras de acceso de índice suelen definirse con base en un solo campo del archivo, el llamado **campo de indización.** Por lo regular, el índice contiene todos los valores del campo de indización junto con una lista de apuntadores a todos los bloques que contienen registros con ese valor en ese campo. Los valores del índice están *ordenados* para que podamos efectuar búsquedas binarias en el índice. Como el archivo del índice es mucho más pequeño que el de datos, una búsqueda binaria en un índice es bastante eficiente. La indización de múltiples niveles hace innecesarias las búsquedas binarias a expensas de la construcción de índices del índice mismo. La indización de múltiples niveles se verá en la sección 5.2.

Hay varios tipos de índices ordenados. Un **índice primario** es un índice especificado sobre el *campo de clave de ordenación* de un archivo de registros ordenados. Recordemos que en la sección 4.7 definimos un campo de clave de ordenación como aquel que sirve para *ordenar físicamente* los registros del archivo en el disco, y que cada registro tiene un *valor único* en ese campo. Si el campo de ordenación no es un campo clave —esto es, si varios registros del archivo pueden tener el mismo valor del campo de ordenación— se puede utilizar otro tipo de índice, el **índice de agrupamiento**. Cabe destacar que un archivo puede tener *cuando más* un campo de ordenación física, así que puede tener cuando más un índice primario o un índice de agrupamiento, *pero no ambos*. Un tercer tipo de índice, el **índice secundario**, se puede especificar sobre cualquier campo del archivo *que no sea el de ordenación*. Un archivo puede tener varios índices secundarios además de su método de acceso primario. En las tres subsecciones que siguen analizaremos estos tres tipos de índices.

5.1.1 *Índices primarios*

Un **índice primario** es un archivo ordenado cuyos registros son de longitud fija y contienen dos campos. El primero de estos campos tiene el mismo tipo de datos que el campo clave de ordenamiento del archivo de datos, y el segundo campo es un apuntador a un bloque de disco: una dirección de bloque. El campo clave de ordenamiento se denomina **clave primaria** del archivo de datos. Hay una **entrada de índice** (o **registro de índice**) en el archivo de índice por cada *bloque* del archivo de datos. Para cada entrada del índice los valores de sus campos son el campo de clave primaria del *primer* registro de un bloque y un apuntador a ese bloque. Denotaremos a estos dos valores de la entrada de índice i con $<K(i), P(i)>$.

Para crear un índice primario del archivo ordenado de la figura 4.7, usamos el campo NOMBRE como clave primaria, porque ése es el campo clave de ordenamiento del archivo (suponiendo que todos los valores de NOMBRE son únicos). Cada entrada del índice tiene un valor de NOMBRE y un apuntador. Las primeras tres entradas del índice son:

$<K(1)$ = (Abad, Adriana), $P(1)$ = dirección del bloque 1$>$
$<K(2)$ = (Acosta, Beatriz), $P(2)$ = dirección del bloque 2$>$
$<K(3)$ = (Aguilera, Héctor), $P(3)$ = dirección del bloque 3$>$

La figura 5.1 ilustra este índice primario. El número total de entradas del índice es igual al *número de bloques de disco* del archivo de datos ordenado. El primer registro de cada bloque

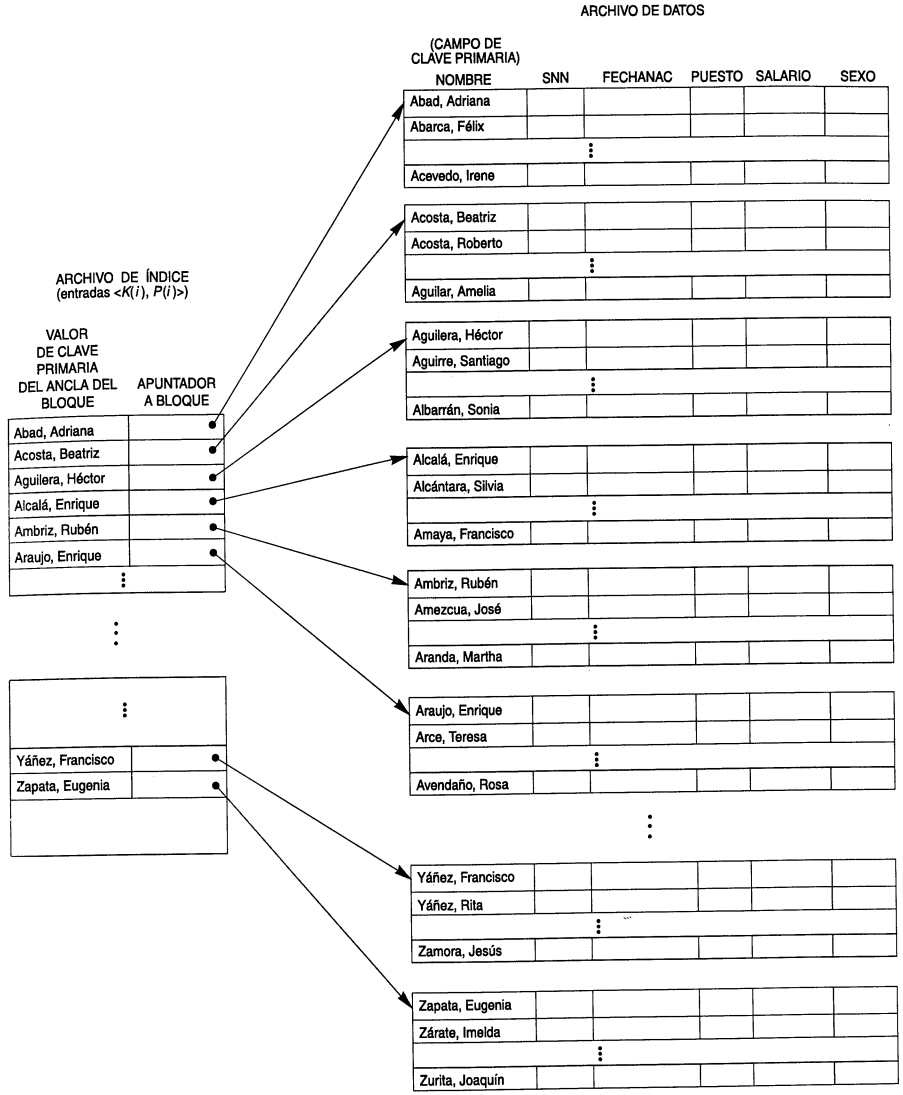


Figura 5.1 Índice primario según el campo de clave de ordenamiento del archivo que se muestra en la figura 4.7.

del archivo de datos se denomina **registro ancla** del bloque, o simplemente **ancla del bloque**.† Los índices primarios son ejemplos de **índices no densos**: cuentan con una entrada por cada bloque de disco del archivo de datos, no por *cada registro* del archivo de datos. Los **índices densos**, en cambio, contienen una entrada por cada registro del archivo de datos.

El archivo de índice de un índice primario requiere muchos menos bloques que el archivo de datos, por dos razones. Primera, hay *menos entradas de índice* que registros en el archivo de datos, porque sólo se necesita una entrada de índice por cada bloque completo del archivo de datos, no por cada registro. Segunda, cada entrada del índice suele ser *de menor tamaño* que un registro de datos porque sólo tiene dos campos; en consecuencia, en un bloque pueden caber más entradas de índice que registros de datos. Es por ello que una búsqueda binaria en el archivo de índice requiere menos accesos a bloques que una búsqueda binaria en el archivo de datos.

Un registro cuyo valor de clave primaria es K está en el bloque cuya dirección es $P(i)$, donde $K(i) \leq K < K(i+1)$. El i-ésimo bloque del archivo de datos contiene todos esos registros debido al ordenamiento físico de los registros del archivo según el campo de clave primaria. Para encontrar un registro, dado el valor K de su campo de clave primaria, realizaremos una búsqueda binaria en el archivo de índice hasta encontrar la entrada de índice apropiada, i, y luego leeremos el bloque del archivo de datos cuya dirección sea $P(i)$. Observe que la fórmula anterior no sería correcta si el archivo de datos estuviera ordenado según un *campo no clave* que pudiera tener valores iguales en varios registros. En tal caso, el mismo valor de índice que está en el ancla del bloque podría estar repetido en los últimos registros del bloque anterior. El ejemplo 1 ilustra el ahorro en accesos a bloques que se puede lograr cuando se utiliza un índice para buscar un registro.

EJEMPLO 1: Suponga que tenemos un archivo ordenado con $r = 30\ 000$ registros almacenados en un disco de tamaño de bloque $B = 1024$ bytes. Los registros del archivo son de longitud fija ($R = 100$ bytes) y no están extendidos. El factor de bloques del archivo sería $fbl = \lfloor (B/R) \rfloor = \lfloor (1024/100) \rfloor = 10$ registros por bloque. El número de bloques requerido para el archivo es $b = \lceil (r/fbl) \rceil = \lceil (30\ 000/10) \rceil = 3000$ bloques. Una búsqueda binaria en el archivo de datos requeriría aproximadamente $\lceil (\log_2 b) \rceil = \lceil (\log_2 3000) \rceil = 12$ accesos a bloques.

Supongamos ahora que el campo clave de ordenación del archivo tiene $V = 9$ bytes de largo, que un apuntador a bloque tiene $P = 6$ bytes de largo y que hemos construido un índice primario para el archivo. El tamaño de cada entrada de índice es $R_i = (9 + 6) = 15$ bytes, de modo que el factor de bloques del índice es $fbl_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entradas por bloque. El número total de entradas del índice, r_i, es igual al número de bloques del archivo de datos, que es 3000. Por tanto, el número de bloques requerido para el índice es $b_i = \lceil (r_i/fbl_i) \rceil = \lceil (3000/68) \rceil = 45$ bloques. Efectuar una búsqueda binaria en el archivo de índice requeriría $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 45) \rceil = 6$ accesos a bloques. Para encontrar el registro en sí empleando el índice necesitaríamos un acceso adicional a un bloque del archivo de datos, para un total de $6 + 1 = 7$ accesos a bloques; bastante mejor que la búsqueda binaria en el archivo de datos, que requiere 12 accesos a bloques. ■

Un problema importante con los índices primarios —y con todos los archivos ordenados— es la inserción y eliminación de registros. En el caso del índice primario el problema se complica porque, si intentamos insertar un registro en su posición correcta dentro del

†Podemos usar un esquema similar al descrito aquí, pero con el último registro de cada bloque (en vez del primero) como ancla del bloque. Esto mejora un poco la eficiencia del algoritmo de búsqueda.

archivo de datos, no sólo debemos desplazar registros a fin de abrir espacio para el nuevo registro, sino que tendremos que modificar algunas entradas del índice, pues el desplazamiento de registros alterará los registros ancla de algunos bloques. Podemos emplear un archivo no ordenado de desborde, como explicamos en la sección 4.7, para reducir este problema. Otra posibilidad es usar una lista enlazada de registros de desborde por cada bloque del archivo de datos. Esto es similar al método del manejo de registros de desborde que describimos en la sección 4.8.2, al hablar de la dispersión. Los registros dentro de cada bloque y su lista enlazada de desborde se pueden ordenar para mejorar el tiempo de obtención. La eliminación de registros se maneja mediante marcadores de eliminación.

5.1.2 *Índices de agrupamiento*

Si los registros de un archivo están ordenados físicamente según un campo no clave que *no tiene un valor distinto para cada registro*, dicho campo se denomina **campo de agrupamiento**. Podemos crear un tipo diferente de índice, llamado **índice de agrupamiento**, para acelerar la obtención de registros que tienen el mismo valor en el campo de agrupamiento. Esto no es lo mismo que un índice primario, en el cual el campo de ordenación del archivo de datos debe tener un *valor distinto* para cada registro.

Un índice de agrupamiento es también un archivo ordenado con dos campos; el primero es del mismo tipo que el campo de agrupamiento del archivo de datos, y el segundo es un apuntador a bloque. Hay una entrada en el índice de agrupamiento por cada *valor distinto* del campo de agrupamiento, y contiene el valor y un apuntador al *primer bloque* del archivo de datos que tiene un registro con ese valor en el campo de agrupamiento. Por ejemplo, la figura 5.2 muestra un archivo de datos con índice de agrupamiento. Observe que la inserción y la eliminación de registros siguen causando problemas, porque los registros de datos están ordenados físicamente. A fin de aliviar el problema de la inserción, se acostumbra reservar un bloque completo por *cada valor* del campo de agrupamiento; todos los registros con ese valor se colocan en el bloque. Si se requiere más de un bloque para almacenar los registros con un valor determinado, se asignan y enlazan bloques adicionales. Esto hace relativamente sencillas la inserción y la eliminación. La figura 5.3 muestra este esquema.

Los índices de agrupamiento son un ejemplo más de índices *no densos*, porque tienen una entrada por cada *valor distinto* del campo de indización, no por cada registro del archivo. Hay una cierta similitud entre las figuras 5.1 y 5.3, por un lado, y las figuras 4.11 y 4.12, por el otro. Los índices son un tanto parecidos a las estructuras de directorio empleadas para la dispersión dinámica y la extensible, que describimos en la sección 4.8.3. En ambas se busca un apuntador al bloque de datos que contiene el registro deseado. Una diferencia importante es que la búsqueda en un índice utiliza los valores del propio campo de búsqueda, en tanto que la búsqueda en un directorio de dispersión emplea los valores de dispersión que se calculan aplicando la función de dispersión al campo de búsqueda.

5.1.3 *Índices secundarios*

Los **índices secundarios** también son archivos ordenados con dos campos. El primero es del mismo tipo que algún *campo no de ordenamiento* del archivo de datos, y se denomina **campo de indización** del mismo. El segundo campo es un apuntador a *bloque* o bien un apuntador a *registro*. Puede haber *muchos* índices secundarios (y, por tanto, campos de indización) para el mismo archivo.

Primero consideraremos una estructura de acceso de índice secundario sobre un campo clave (uno que tiene un *valor distinto* para cada registro del archivo de datos). En ocasiones a estos campos se les llama **claves secundarias.** En este caso hay una entrada de índice por *cada registro* del archivo de datos, y contiene el valor de la clave secundaria para ese registro y un apuntador, ya sea al bloque en el que está almacenado ese registro o al registro mismo. Los índices secundarios según campos clave son índices **densos**: contienen una entrada por cada registro del archivo.

Una vez más, nos referimos a los dos valores de la entrada de índice i como $<K(i), P(i)>$. Las entradas están **ordenadas** según el valor de $K(i)$, así que podemos realizar una

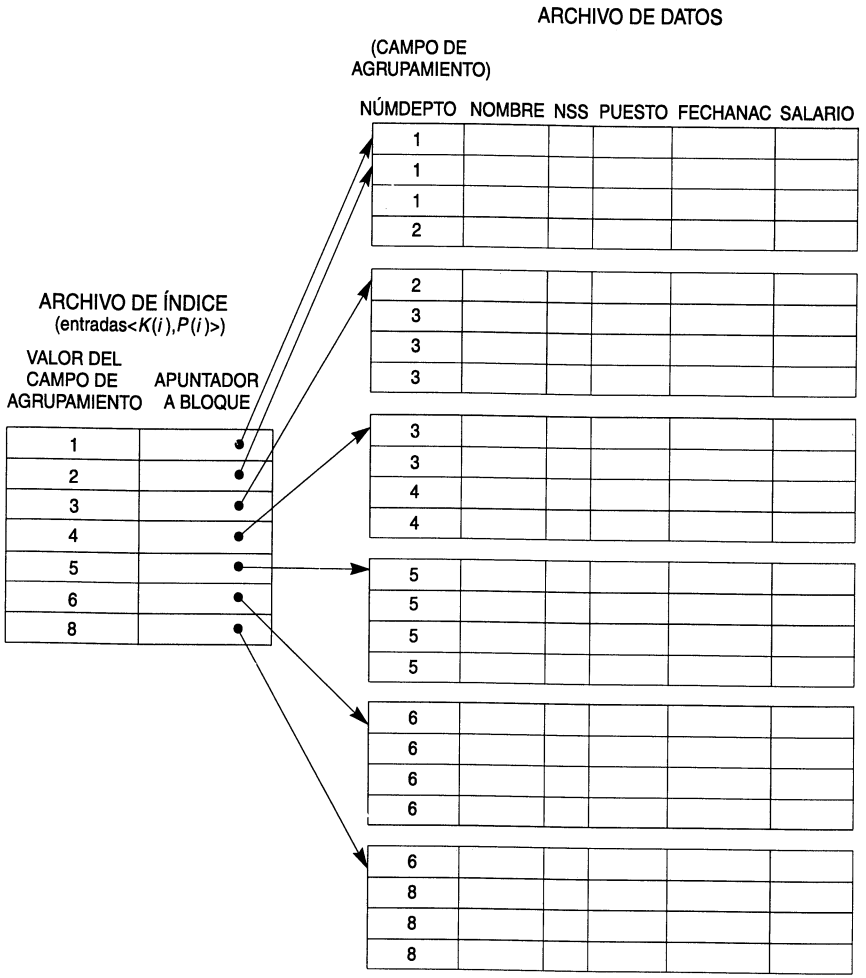


Figura 5.2 Índice de agrupamiento según el campo de ordenamiento NÚMDEPTO de un archivo EMPLEADO.

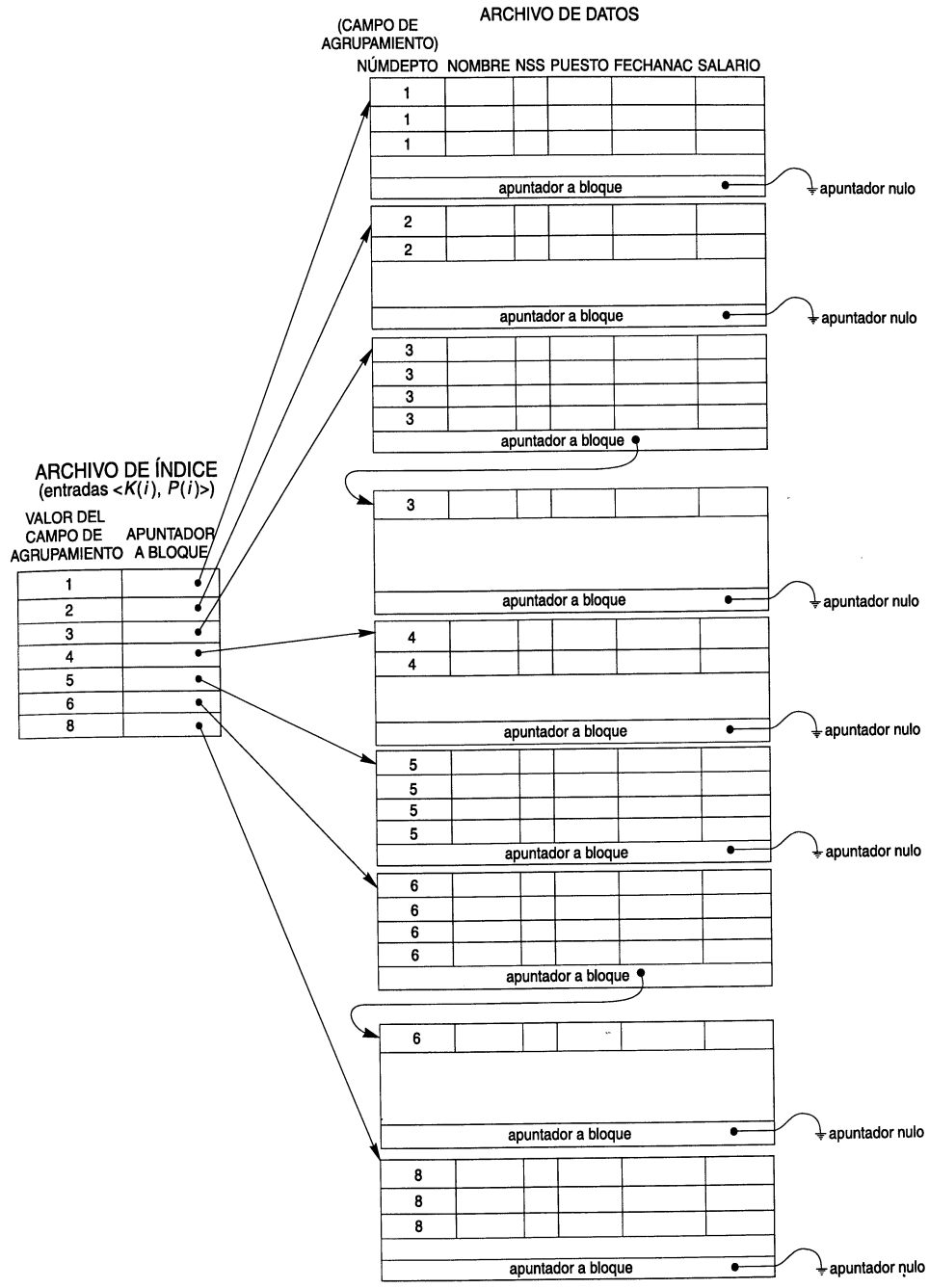


Figura 5.3 Índice de agrupamiento con bloques individuales para cada grupo de registros que comparten el mismo valor del campo de agrupamiento.

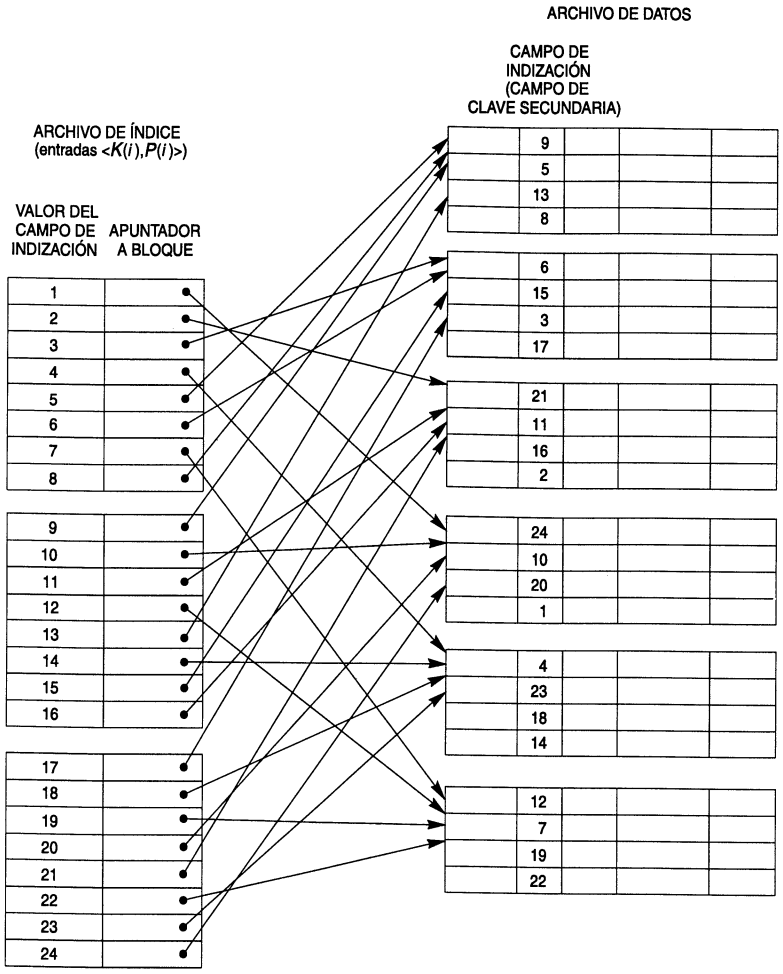


Figura 5.4 Índice secundario denso según un campo clave que no determina el ordenamiento del archivo.

búsqueda binaria en el índice. Como los registros del archivo de datos *no* están ordenados según los valores del campo de clave secundaria, no podemos usar anclas de bloques. Es por ello que se crea una entrada de índice por cada registro del archivo de datos y no por cada bloque, como en el caso de los índices primarios. La figura 5.4 ilustra un índice secundario en el que los apuntadores $P(i)$ de las entradas del índice son *apuntadores a bloques*, no apuntadores a registros. Una vez transferido el bloque apropiado a la memoria principal, se puede efectuar una búsqueda del registro deseado dentro del bloque.

Por lo regular, los índices secundarios necesitan más espacio de almacenamiento y tiempos de búsqueda más largos que los primarios, debido a su mayor número de entradas. No obstante, la *mejoría* en el tiempo de búsqueda de un registro arbitrario es mucho mayor para un índice secundario que para uno primario, pues tendríamos que realizar una *búsqueda lineal* en el archivo de datos si no existiera el índice secundario. En el caso de un índice primario, podríamos realizar una búsqueda binaria en el archivo principal, incluso si no existiera dicho índice. El ejemplo 2 ilustra la mejoría en el número de accesos a bloques requeridos cuando se utiliza un índice secundario para buscar un registro.

EJEMPLO 2: Consideremos el archivo del ejemplo 1 con $r = 30\ 000$ registros de longitud fija de tamaño $R = 100$ bytes, almacenado en un disco cuyos bloques son de $B = 1024$ bytes. El archivo tiene $b = 3000$ bloques, según se calculó en el ejemplo 1. Para efectuar una búsqueda lineal en este archivo tendríamos que examinar $b/2 = 3000/2 = 1500$ accesos a bloques en promedio. Supongamos que construimos un índice secundario basado en un campo clave (no de ordenamiento) del archivo que tiene $V = 9$ bytes de longitud. Al igual que en el ejemplo 1, un apuntador a bloque tiene 6 bytes de largo, así que cada entrada de índice tiene $R_i = (9 + 6) = 15$ bytes, y el factor de bloques del índice tiene $fbl_i = \lfloor (B/R_i) \rfloor = \lfloor (1024/15) \rfloor = 68$ entradas por bloque. En un índice secundario denso como éste, el número total de entradas de índice, r_i, es igual al *número de registros* del archivo de datos, que es 30 000. El número de bloques requeridos para el índice será entonces $b_i = \lceil (r_i/fbl_i) \rceil = \lceil (30\ 000/68) \rceil = 442$ bloques. Compárese esto con los 45 bloques que necesita el índice primario no denso del ejemplo 1.

Una búsqueda binaria en este índice secundario requiere $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$ accesos a bloques. Para encontrar el registro empleando el índice, requeriremos un acceso adicional a un bloque del archivo de datos para un total de $9 + 1 = 10$ accesos a bloques; mucho mejor que los 1500 accesos a bloques que requiere en promedio una búsqueda lineal en este archivo, pero no tan bueno como los 7 accesos a bloques que se necesitan con el índice primario. ■

También podemos crear un índice secundario con base en un *campo no clave* de un archivo. En este caso, varios registros del archivo de datos pueden tener el mismo valor en el campo de indización. Disponemos de varias opciones para implementar un índice así:

- La opción 1 es incluir varias entradas de índice con el mismo valor $K(i)$, una por registro. El índice sería denso.
- La opción 2 es usar registros de longitud variable para las entradas del índice, con un campo repetitivo para el apuntador. Mantendremos una lista de apuntadores $<P(i,1), ..., P(i,k)>$ en la entrada de índice de $K(i)$: un apuntador a cada bloque que contenga un registro cuyo valor del campo de indización sea igual a $K(i)$. Tanto en la opción 1 como en la 2 será necesario modificar de manera apropiada el algoritmo para efectuar una búsqueda binaria en el índice.
- La opción 3, que se utiliza más a menudo, es usar entradas de índice de longitud fija y tener una sola entrada por cada *valor del campo de indización*, pero creando un nivel de indirección adicional para manejar los apuntadores múltiples. En este esquema no denso, el apuntador $P(i)$ de una entrada de índice $<K(i), P(i)>$ apunta a un *bloque de apuntadores a registros*; cada apuntador a registro de ese bloque apunta a uno de los registros del archivo de datos que tienen el valor $K(i)$ en el campo de indización. Si algún valor $K(i)$ ocurre en demasiados registros, de modo que sus apuntadores a registros no quepan en un solo bloque de disco, se utiliza una lista enlazada de bloques. Esta

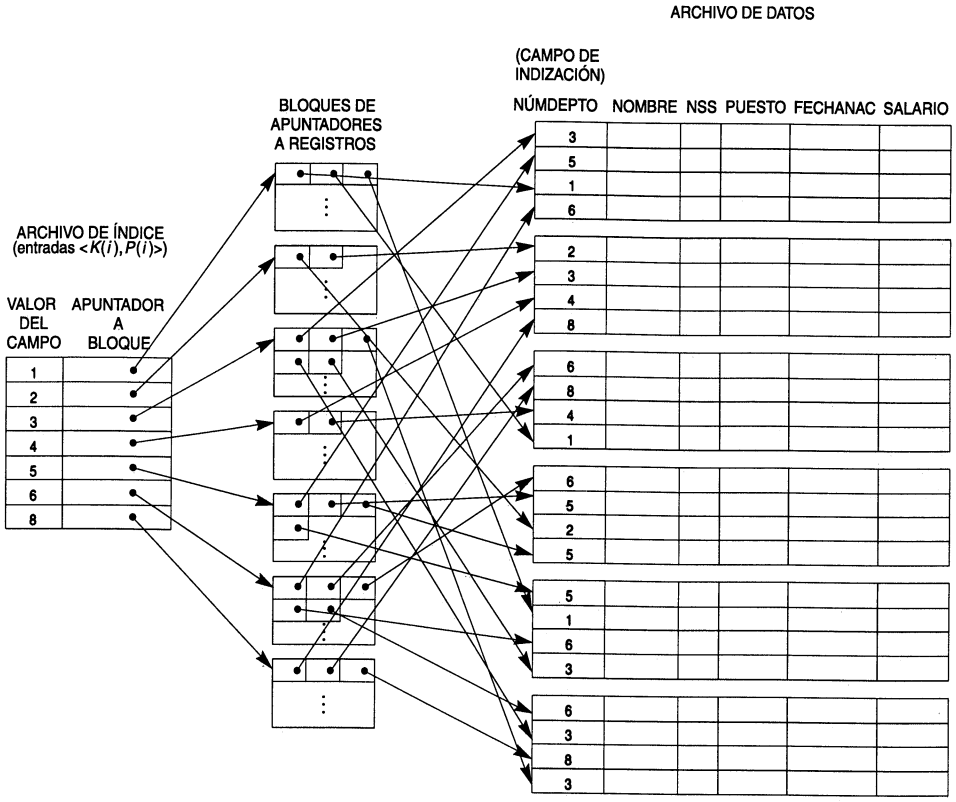


Figura 5.5 Índice secundario según un campo no clave, implementado con un nivel de indirección para que las entradas del índice tengan longitud fija y valores únicos en los campos.

técnica se ilustra en la figura 5.5. La obtención de datos a través del índice requiere un acceso a bloque adicional debido al nivel extra, pero los algoritmos para buscar en el índice y (lo que es más importante) para insertar nuevos registros en el archivo de datos son sencillos. Además, las búsquedas con condiciones de selección complejas pueden manejarse haciendo referencia a los apuntadores, sin tener que leer muchos registros innecesarios del archivo (véase el ejercicio 5.16).

Observe que los índices secundarios representan un **ordenamiento lógico** de los registros según el campo de indización. Si tenemos acceso a los registros en el orden que tienen las entradas del índice secundario, los leeremos en orden según el campo de indización.

5.1.4 *Resumen*

Para concluir esta sección, sintetizamos en dos tablas nuestro análisis de los tipos de índices. La tabla 5.1 muestra las características que tiene el campo de indización de todos los

Tabla 5.1 Tipos de índices

	Campo de ordenación	Campo no de ordenación
Campo clave	Índice primario	Índice secundario (clave)
Campo no clave	Índice de agrupamiento	Índice secundario (no clave)

Tabla 5.2 Propiedades de los tipos de índices

		Propiedades del tipo de índices		
		Número de entradas de índice (primer nivel)	**Denso o no denso**	**Anclas de bloques del archivo de datos**
Tipo de índice	Primario	Número de bloques del archivo de datos	No denso	Sí
	De agrupamiento	Número de valores distintos del campo índice	No denso	Sí/no[a]
	Secundario (clave)	Número de registros del archivo de datos	Denso	No
	Secundario (no clave)	Número de registros[b] o número de valores distintos del campo índice[c]	Denso o no denso	No

[a]Sí en el caso de que cada valor distinto del campo de ordenación inicie un nuevo bloque; no en caso contrario.
[b]Para la opción 1.
[c]Para las opciones 2 y 3.

tipos de índices ordenados de un solo nivel que hemos visto: primarios, de agrupamiento y secundarios. La tabla 5.2 resume las propiedades de cada tipo de índices comparando el número de entradas del índice y especificando cuáles índices son densos y cuáles emplean anclas de bloque en el archivo de datos.

5.2 Índices de múltiples niveles

Los esquemas de indización hasta aquí descritos implican un archivo de índice ordenado. Aplicamos una búsqueda binaria al índice para localizar apuntadores a los registros del archivo que tienen un cierto valor del campo de indización. Una búsqueda binaria requiere aproximadamente $(\log_2 b_i)$ accesos a bloques en el caso de un índice con b_i bloques, porque cada paso del algoritmo reduce a la mitad la porción del archivo de índice que seguiremos examinando; por tanto aplicamos la función logarítmica de base 2. Los **índices de múltiples niveles** se basan en la idea de reducir en fbl_i la porción del índice que seguiremos examinando, donde fbl_i es el factor de bloques del índice y es mayor que 2. Por tanto, el espacio de búsqueda se reduce con mucha rapidez. El valor fbl_i se conoce como **abanico** (*fan-out*) del índice de múltiples niveles, y nos referiremos a él con la abreviatura **fo**. Una búsqueda en un índice de múltiples niveles requiere aproximadamente $(\log_{fo} b_i)$ accesos a bloques, que es una cifra menor que la de la búsqueda binaria si el abanico es mayor que 2.

El índice de múltiples niveles considera al archivo de índice, al que ahora llamaremos **primer nivel** (o **nivel base**) del índice de múltiples niveles, como un *archivo ordenado* con un *valor distinto* para cada $K(i)$. Por tanto, podemos crear un índice primario para este primer nivel; este índice del primer nivel es ahora el **segundo nivel** del índice de múltiples niveles. Como el segundo nivel es un índice primario, podemos usar anclas de bloques para que el segundo nivel tenga una entrada *por cada bloque* del primer nivel. El factor de bloques

fbl_i del segundo nivel —y de todos los niveles subsecuentes— es el mismo que el del índice de primer nivel, porque todas las entradas de índice tienen el mismo tamaño; cada una tiene un valor de campo y una dirección de bloque. Si el primer nivel tiene r_1 entradas, y el factor de bloques —que es también el abanico— del índice es $fbl_i = fo$, entonces el primer nivel requerirá $\lceil (r_1/fo) \rceil$ bloques, que en consecuencia será el número de entradas r_2 requeridas en el segundo nivel del índice.

Podemos repetir este proceso para el segundo nivel. El **tercer nivel**, que es un índice primario del segundo nivel, tiene una entrada por cada bloque del segundo nivel, así que el número de entradas del tercer nivel es $r_3 = \lceil (r_2/fo) \rceil$. Cabe señalar que sólo necesitaremos un segundo nivel si el primero requiere más de un bloque de almacenamiento en disco, y, de manera similar, sólo necesitaremos un tercer nivel si el segundo requiere más de un bloque. Podemos repetir el proceso anterior hasta que todas las entradas de un nivel t del índice quepan en un solo bloque. Este bloque del t-ésimo nivel se denomina índice de nivel **superior.**[†] Cada nivel reduce el número de entradas del nivel anterior en un factor de fo (el abanico del índice), así que podemos usar la fórmula $1 \leq (r_1/((fo)^t))$ para calcular t. Por tanto, un índice de múltiples niveles con r_1 entradas en el primer nivel tendrá aproximadamente t niveles, donde $t = \lceil (\log_{fo}(r_1)) \rceil$.

El esquema de múltiples niveles que hemos descrito es útil para cualquier tipo de índice, sea primario, de agrupamiento o secundario, en tanto el índice del primer nivel tenga *valores distintos para K(i) y entradas de longitud fija*. La figura 5.6 muestra un índice de múltiples niveles construido sobre un índice primario. El ejemplo 3 ilustra la mejoría en el número de bloques leídos cuando se utiliza un índice de múltiples niveles para buscar un registro.

EJEMPLO 3: Suponga que el índice secundario denso del ejemplo 2 se convierte en un índice de múltiples niveles. Calculamos un factor de bloques $fbl_i = 68$ entradas de índice por bloque, cifra que también es el abanico fo para el índice de múltiples niveles. También se calculó el número de bloques del primer nivel $b_1 = 442$ bloques. El número de bloques del segundo nivel será $b_2 = \lceil (b_1/fo) \rceil = \lceil (442/68) \rceil = 7$ bloques, y el número de bloques del tercer nivel será $b_3 = \lceil (b_2/fo) \rceil = \lceil (7/68) \rceil = 1$ bloque. Por tanto, el tercer nivel es el nivel superior del índice, y $t = 3$. Para tener acceso a un registro buscando en el índice de múltiples niveles, deberemos tener acceso a un bloque en cada nivel y a uno más en el archivo de datos, así que necesitamos $t + 1 = 3 + 1 = 4$ accesos a bloques. Compárese esto con el ejemplo 2, donde se requirieron 10 accesos a bloques empleando un índice de un solo nivel y una búsqueda binaria. ■

Cabe señalar que también podríamos tener un índice primario de múltiples niveles, que sería no denso. El ejercicio 5.10c ilustra este caso, donde *es preciso* tener acceso al bloque de datos del archivo antes de que podamos determinar si el registro que se busca está o no en el archivo. En el caso de un índice denso, esto puede determinarse teniendo acceso al primer nivel del índice (sin tener que leer un bloque de datos), ya que hay una entrada de índice *por cada* registro del archivo.

El algoritmo 5.1 bosqueja el procedimiento de búsqueda de un registro en un archivo de datos que utiliza un índice primario de t niveles. Nos referiremos a la entrada i del nivel

[†]El esquema de numeración de los niveles del índice que usamos aquí es el *inverso* de la forma como suelen definirse los niveles de las estructuras de datos de árbol. En ellas, t se considera el nivel 0 (cero), $t - 1$ es el nivel 1, etcétera.

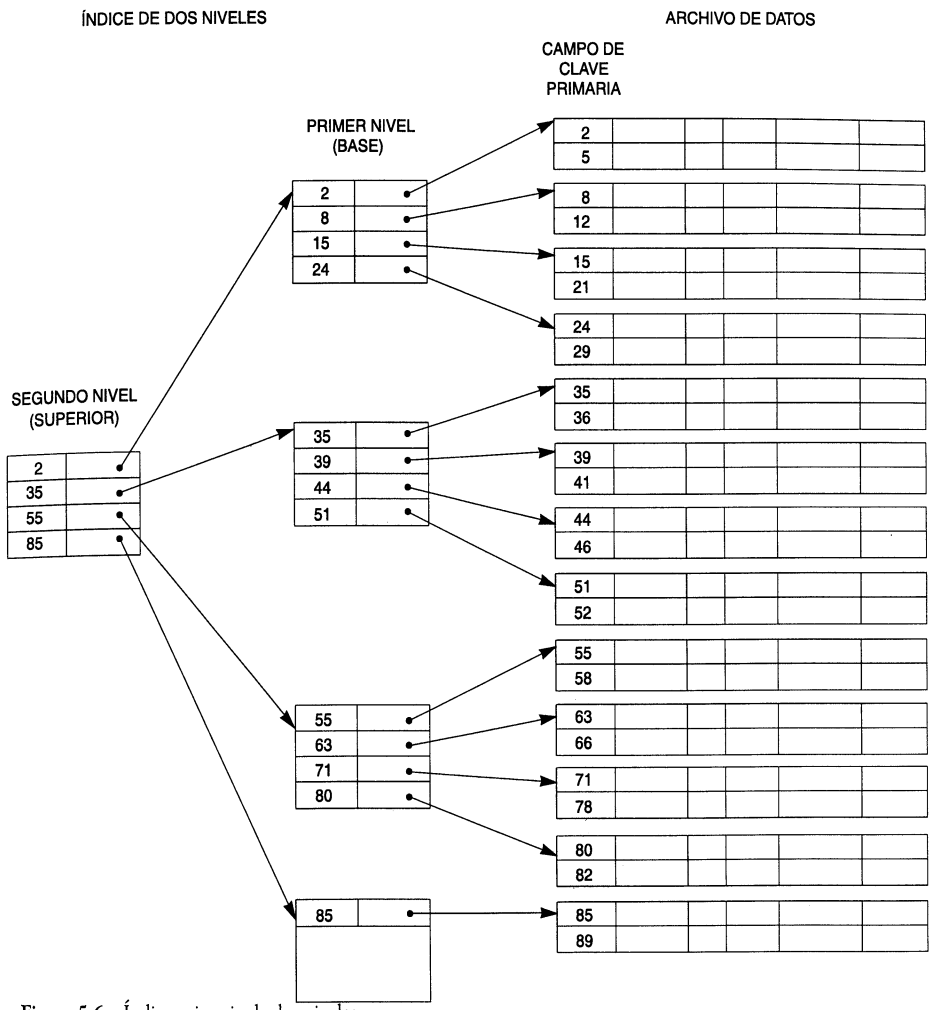


Figura 5.6 Índice primario de dos niveles.

j del índice como $<K_j(i), P_j(i)>$, y buscaremos un registro cuyo valor de clave primaria sea K. Supondremos que se ignoran cualesquiera registros de desborde. Si el registro está en el archivo, deberá haber una entrada en el nivel 1 con $K_1(i) \leq K < K_1(i + 1)$, y el registro estará en el bloque del archivo de datos cuya dirección sea $P_1(i)$. En el ejercicio 5.15 se analiza la modificación del algoritmo de búsqueda para otros tipos de índices.

ALGORITMO 5.1 Búsqueda en un índice primario no denso de t niveles.

```
p ← dirección del bloque de nivel superior del índice;
para j ← t paso –1 a 1 hacer
  comenzar
  leer el bloque de índice (en el j-ésimo nivel) cuya dirección es p;
```

buscar en el bloque p la entrada i tal que $K_j(i) \leq K < K_j(i+1)$ (si $K_j(i)$ es la última entrada del bloque, basta con satisfacer $K_j(i) \leq K$);
$p \leftarrow P_j(i)$ (* escoge el apuntador apropiado en el j-ésimo nivel del índice *)
terminar;
leer el bloque del archivo de datos cuya dirección es p;
buscar en el bloque p el registro de clave = K;

Como hemos visto, los índices de múltiples niveles reducen el número de bloques leídos al buscar un registro, dado su valor del campo de indización. Falta resolver los problemas de insertar y eliminar entradas del índice, pues todos los niveles del índice son *archivos ordenados físicamente*. A fin de seguir aprovechando la indización de múltiples niveles y a la vez reducir los problemas de inserción y eliminación en el índice, los diseñadores a menudo adoptan un índice de múltiples niveles que deja cierto espacio en todos sus bloques para insertar nuevas entradas. Esto se conoce como **índice dinámico de múltiples niveles**, y a menudo se implementa con las estructuras de datos llamadas árboles B y árboles B+.

5.3 Índices dinámicos de múltiples niveles con base en árboles B y B+

Los árboles B y B+ son casos especiales de las estructuras de árbol, tan conocidas. Presentaremos brevemente la terminología empleada al hablar de las estructuras de datos de árbol. Un **árbol** se compone de **nodos**. Cada nodo del árbol, con excepción de un nodo especial llamado **raíz**, tiene un nodo **padre** y varios —cero o más— nodos **hijos**. El nodo raíz no tiene padre. Los nodos que no tienen hijos se denominan nodos **hoja**, y los nodos que no son hojas se conocen como nodos **internos**. El **nivel** de un nodo es siempre el nivel de su padre más uno, y el nivel del nodo raíz es cero.† Un **subárbol** de un nodo consiste en ese nodo y todos sus

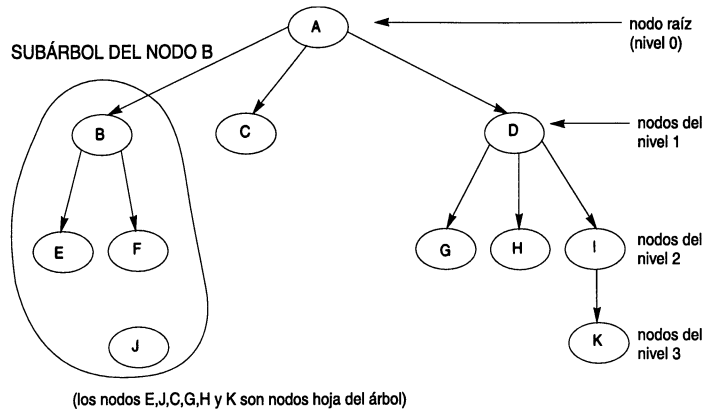


Figura 5.7 Estructura de datos de árbol.

†Esta definición estándar del nivel de un nodo, que adoptaremos en toda la sección 5.3, es diferente de la que dimos para los índices de múltiples niveles en la sección 5.2.

nodos **descendientes**: sus nodos hijo, los nodos hijo de sus nodos hijo, etc. Una definición recursiva precisa de subárbol es que consiste en un nodo n y en los subárboles de los nodos hijo de n. La figura 5.7 ilustra una estructura de datos de árbol. En ella, el nodo raíz es A y sus nodos hijo son B, C y D. Los nodos E, J, C, G, H y K son nodos hoja.

Por lo regular, mostramos los árboles con el nodo raíz en la parte superior, como en la figura 5.7. Una forma de implementar un árbol es tener tantos apuntadores en un nodo como nodos hijo tenga ese nodo. En algunos casos, también se almacena un apuntador al padre. Además de apuntadores, los nodos suelen contener información almacenada. Cuando un índice de múltiples niveles se implementa en forma de estructura de árbol, dicha información incluye los valores del campo de indización del archivo que sirven para guiar la búsqueda de un registro en particular.

En la sección 5.3.1 presentaremos los árboles de búsqueda y luego hablaremos de los árboles B, que pueden servir como índices dinámicos de múltiples niveles para guiar la búsqueda de registros en un archivo de datos. Los nodos de un árbol B se mantienen ocupados entre el 50% y el 100%, y los apuntadores a los bloques de datos se almacenan tanto en los nodos internos como en los nodos hoja de la estructura de árbol B. En la sección 5.3.2 estudiaremos los árboles B+, una variación de los árboles B, en los que sólo se almacenan apuntadores a los bloques de datos del archivo en los nodos hoja; esto permite lograr índices con menos niveles y de más alta capacidad.

5.3.1 *Árboles de búsqueda y árboles B*

Un árbol de búsqueda es un tipo especial de árbol que sirve para guiar la búsqueda de un registro, dado el valor de uno de sus campos. Los índices de múltiples niveles analizados en la sección 5.2 pueden considerarse como variaciones de los árboles de búsqueda. Cada nodo del índice de múltiples niveles puede tener hasta fo apuntadores y fo valores de clave, donde fo es el abanico (*fan-out*) del índice. Los valores del campo de índice de cada nodo nos guían al siguiente nodo, hasta llegar al bloque del archivo de datos que contiene los registros deseados. Al seguir un apuntador, restringimos nuestra búsqueda en cada nivel a un subárbol del árbol de búsqueda e ignoramos todos los nodos que no estén en dicho subárbol.

Árboles de búsqueda. Los árboles de búsqueda difieren un poco de los índices de múltiples niveles. Un **árbol de búsqueda** de orden p es un árbol tal que cada nodo contiene *cuando más* $p-1$ valores de búsqueda y p apuntadores en el orden $<P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q>$, donde $q \leq p$, cada P_i es un apuntador a un nodo hijo (o un apuntador nulo) y cada K_i es un valor de búsqueda proveniente de algún conjunto ordenado de valores. Se supone que todos los

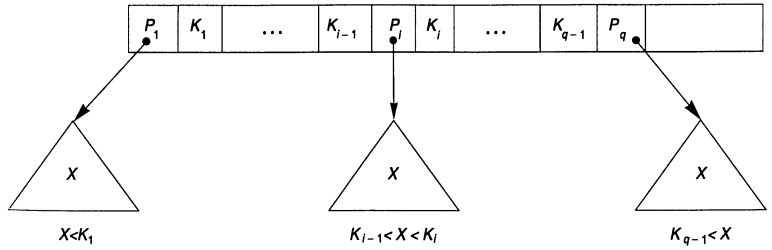


Figura 5.8 Nodo de un árbol de búsqueda.

valores de búsqueda son únicos.† La figura 5.8 ilustra un nodo de un árbol de búsqueda. Este último debe ajustarse en todo momento a las dos restricciones siguientes:

1. Dentro de cada nodo, $K_1 < K_2 < ... < K_{q-1}$.
2. Para todos los valores de X del subárbol al cual apunta P_i, tenemos $K_{i-1} < X < K_i$, para $1 < i < q$, $X < K_i$, para $i = 1$, y $K_{i-1} < X$, para $i = q$ (Fig. 5.8).

Al buscar un valor X, siempre seguimos el apuntador P_i apropiado de acuerdo con las fórmulas de la restricción 2. La figura 5.9 ilustra un árbol de búsqueda de orden $p = 3$ con valores de búsqueda enteros. Observe que algunos de los apuntadores P_i de un nodo pueden ser apuntadores **nulos**.

Podemos usar un árbol de búsqueda como mecanismo para buscar registros almacenados en un archivo de disco. Los valores del árbol pueden ser los valores de uno de los campos del archivo, el llamado **campo de búsqueda** (igual al campo de indización si un índice de múltiples niveles guía la búsqueda). Cada valor del árbol está asociado a un apuntador al registro que tiene ese valor en el archivo de datos. Como alternativa, puede apuntar al bloque de disco que contiene ese registro. El árbol de búsqueda en sí puede almacenarse en disco asignando cada nodo del árbol a un bloque del disco. Cuando se inserta un registro nuevo, es preciso actualizar el árbol de búsqueda incluyendo en él el valor del campo de búsqueda del nuevo registro y un apuntador a éste.

Se requieren algoritmos para insertar valores de búsqueda en el árbol y eliminarlos de él sin violar las dos restricciones mencionadas. En general, estos algoritmos no garantizan que el árbol de búsqueda esté **equilibrado**; esto es, que todos sus nodos hoja estén en el mismo nivel.†† Es importante mantener equilibrados los árboles de búsqueda porque esto garantiza que no habrá nodos en niveles muy altos que requieran muchos accesos a bloques durante una búsqueda. Otro problema de los árboles de búsqueda es que la eliminación de registros puede dejar algunos nodos del árbol casi vacíos, desperdiciándose espacio de almacenamiento y aumentándose el número de niveles.

Árboles B. El árbol B —un árbol de búsqueda con algunas restricciones adicionales— resuelve hasta cierto punto los dos problemas anteriores. Dichas restricciones adicionales garantizan que el árbol siempre estará equilibrado y que el espacio desperdiciado por la eliminación, si lo hay, nunca será excesivo. Los algoritmos para insertar y eliminar, empero, aumentan en complejidad para poder mantener estas restricciones. No obstante, la mayor parte de las inserciones y eliminaciones son procesos simples; se complican sólo en circunstancias especiales: a saber, cuando intentamos una inserción en un nodo que ya está lleno o una eliminación en un nodo que después quedará ocupado hasta menos de la mitad. En términos más formales, un **árbol B** de **orden** p, utilizado como estructura de acceso según un *campo clave* para buscar registros de un archivo de datos, puede definirse como sigue:

1. Cada nodo interno del árbol B (Fig. 5.10(a)) tiene la forma

$$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_q>$$

donde $q \leq p$. Cada P_i es un **apuntador de árbol**: un apuntador a otro nodo del árbol B. Cada Pr_i es un **apuntador de datos**:††† un apuntador al registro cuyo valor del campo clave de búsqueda es igual a K_i (o un apuntador al bloque que contiene ese registro en el archivo de datos).

2. Dentro de cada nodo, $K_1 < K_2 < \cdots < K_{q-1}$.
3. Para todos los valores X del campo clave de búsqueda en el subárbol al que apunta P_i, tenemos

$$K_{i-1} < X < K_i, \text{ para } 1 < i < q,\ X < K_i, \text{ para } i = 1, \text{ y } K_{i-1} < X, \text{ para } i = q$$

(véase la Fig. 5.10(a)).

4. Cada nodo tiene cuando más p apuntadores de árbol.
5. Cada nodo, excepto la raíz y los nodos hoja, tienen por lo menos $\lceil (p/2) \rceil$ apuntadores de árbol. El nodo raíz tiene como mínimo dos apuntadores de árbol, a menos que sea el único nodo del árbol.
6. Un nodo con q apuntadores de árbol, $q \leq p$, tiene $q - 1$ valores del campo clave de búsqueda (y por tanto tiene $q - 1$ apuntadores de datos).
7. Todos los nodos hoja están en el mismo nivel. Los nodos hoja tienen la misma estructura que los internos, excepto que todos sus *apuntadores de árbol*, P_i, son **nulos**.

La figura 5.10(b) ilustra un árbol B de orden $p = 3$. Observe que todos los valores de búsqueda K del árbol B son únicos porque supusimos que el árbol se utiliza como estructura de acceso según un campo clave. Si usamos un árbol B con un campo no clave, tendremos que modificar la definición de los apuntadores de datos Pr_i de modo que apunten a un bloque —o a una lista enlazada de bloques— que contenga apuntadores a los registros mismos del archivo. Este nivel adicional de indirección es similar a la opción 3 de los índices secundarios, que vimos en la sección 5.1.3.

Todo árbol B comienza con un solo nodo raíz (que también es un nodo hoja) en el nivel 0 (cero). Una vez lleno este nodo con $p - 1$ valores de la clave de búsqueda, si intentamos insertar una entrada más en el árbol, el nodo raíz se dividirá en dos nodos de nivel 1. En el nodo raíz sólo se mantendrá el valor de en medio: los demás se dividirán equitativamente entre los otros dos nodos. Cuando se llena un nodo distinto de la raíz y se inserta en él una nueva entrada, el nodo se divide en dos nodos del mismo nivel y la entrada de en medio se

†Esta restricción se puede hacer menos rigurosa, pero entonces habría que modificar las fórmulas que siguen.

††La definición de *equilibrado* es diferente en el caso de los árboles binarios. Los árboles binarios equilibrados se conocen como árboles AVL.

†††Un apuntador de datos es una dirección de bloque o bien una dirección de registro; la segunda casi siempre es una dirección de bloque y un desplazamiento.

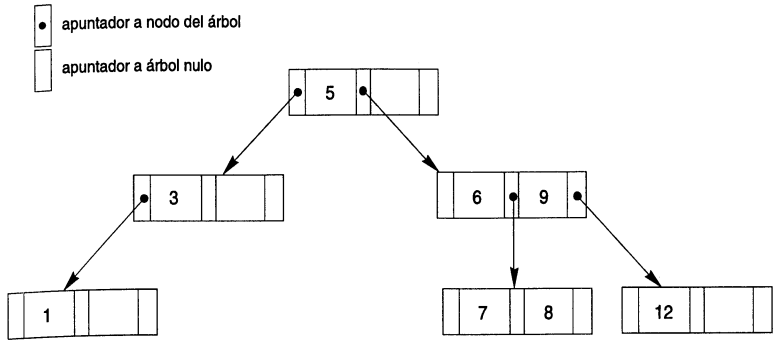


Figura 5.9 Árbol de búsqueda de orden $p = 3$.

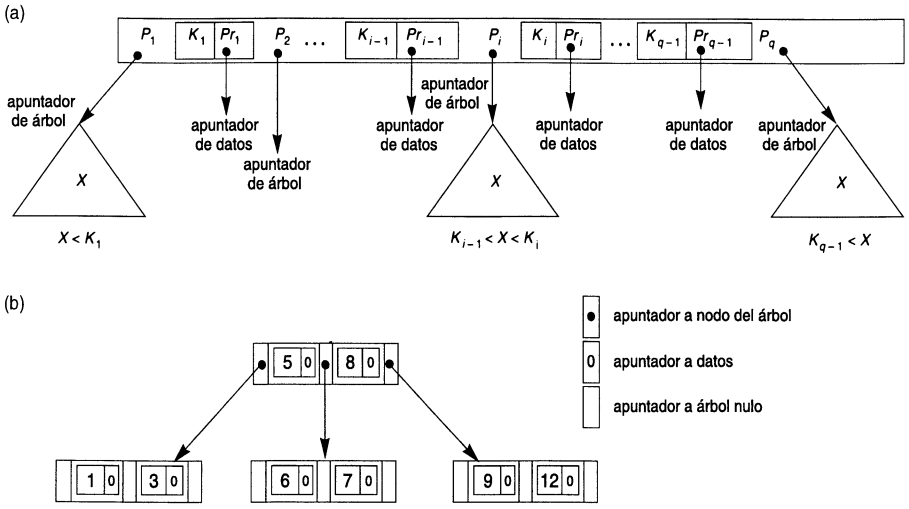


Figura 5.10 Estructuras de árbol B. (a) Nodo de árbol B con $q - 1$ valores de búsqueda. (b) Árbol B de orden $p = 3$. Los valores se insertaron en el orden 8, 5, 1, 7, 3, 12, 9, 6.

pasa al nodo padre junto con dos apuntadores a los nodos divididos. Si el nodo padre está lleno, también se divide. La división puede propagarse hasta el nodo raíz, creando un nuevo nivel cada vez que se divida la raíz. No analizaremos con detalle aquí los algoritmos de los árboles B; más bien, bosquejaremos procedimientos de búsqueda y de inserción para los árboles B^+ en la siguiente sección.

Si la eliminación de un valor hace que un nodo quede ocupado hasta menos de la mitad, se combinará con sus nodos vecinos, y esto también puede propagarse hasta la raíz; por tanto, la eliminación puede reducir el número de niveles del árbol. Se ha demostrado por análisis y simulación que, después de un gran número de inserciones y eliminaciones aleatorias en un árbol B, los nodos están ocupados en un 69%, aproximadamente, cuando se estabiliza el número de valores del árbol. Esto también es verdadero en el caso de los árboles B+. Si llega a suceder esto, la división y combinación de nodos ocurrirá con muy poca frecuencia, de modo que la inserción y la eliminación se volverán muy eficientes. Si crece el número de valores el árbol se expandirá sin problemas, aunque es posible que haya división de nodos, con lo que algunas inserciones tardarán más. El ejemplo 4 ilustra la forma en que calculamos el orden p de un árbol B almacenado en disco.

EJEMPLO 4: Supongamos que el campo de búsqueda tiene $V = 9$ bytes de largo, que el tamaño de un bloque de disco es $B = 512$ bytes, que un apuntador de registro (de datos) tiene $P_r = 7$ bytes y que un apuntador de bloque tiene $P = 6$ bytes. Cada nodo del árbol B puede tener *cuando más* p apuntadores de árbol, $p - 1$ apuntadores de datos y $p - 1$ valores del campo clave de búsqueda (véase la Fig. 5.10(a)). Éstos deben caber en un solo bloque de disco si queremos que cada nodo del árbol corresponda a un bloque de disco. Por tanto, debemos tener

$$(p * P) + ((p - 1) * (P_r + V)) \leq B$$
$$(p * 6) + ((p - 1) * (7 + 9)) \leq 512$$
$$(22 * p) \leq 528$$

En general, un nodo de árbol B puede contener la información adicional requerida por los algoritmos que manipulan el árbol, como el número de entradas q del nodo y un apuntador al nodo padre. Por tanto, antes de efectuar el cálculo anterior para p, tendremos que reducir el tamaño del bloque en la cantidad de espacio necesario para dicha información. Ahora ilustraremos la forma de calcular el número de bloques y niveles de un árbol B.

EJEMPLO 5: Supongamos que el campo de búsqueda del ejemplo 4 es un campo clave que no es el de ordenación, y que construimos un árbol B sobre ese campo. Supongamos también que todos los nodos del árbol están ocupados al 69% de su capacidad. Cada nodo, en promedio, tendrá $p * 0.69 = 23 * 0.69$, o aproximadamente 16 apuntadores y, por ende, 15 valores del campo clave de búsqueda. El **abanico medio** $fo = 16$. Podemos comenzar en la raíz y ver cuántos valores y apuntadores hay, en promedio, en cada nivel subsecuente:

Raíz:	1 nodo	15 entradas	16 apuntadores
Nivel 1:	16 nodos	240 entradas	256 apuntadores
Nivel 2:	256 nodos	3840 entradas	4096 apuntadores
Nivel 3:	4096 nodos	61 440 entradas	

En cada nivel calculamos el número de entradas multiplicando el número total de apuntadores del nivel anterior por 15, el promedio de entradas en cada nodo. Por tanto, para los tamaños de bloque, de apuntador y de campo de clave de búsqueda dados, un árbol B de dos niveles contiene hasta 3840 + 240 + 15 = 4095 entradas en promedio; un árbol B de tres niveles contiene hasta 65 535 entradas en promedio. ■

A veces se emplean árboles B como organizaciones primarias de los archivos, en cuyo caso se almacenan registros completos en los nodos del árbol B, no sólo las entradas <clave de búsqueda, apuntador a registro>. Esto funciona correctamente si el archivo tiene un *número relativamente pequeño de registros* y *los registros son pequeños*. En caso contrario, el abanico y el número de niveles se incrementan tanto que impiden un acceso eficiente.

5.3.2 *Árboles B+*

En su mayoría, las implementaciones de índices dinámicos de múltiples niveles emplean una variación de la estructura de datos de árbol B: el **árbol B+**. En un árbol B, todos los valores del campo de búsqueda aparecen una vez en algún nivel del árbol, junto con un apuntador de datos. En un árbol B+, los apuntadores de datos se almacenan *sólo en los nodo hoja* del árbol, por lo cual la estructura de los nodos hoja difiere de la de los nodos internos. Los primeros tienen una entrada por *cada* valor del campo de búsqueda, junto con un apuntador de datos al registro (o al bloque que contiene el registro), si el campo de búsqueda es un campo clave. Si no lo es, el apuntador apunta a un bloque que contiene apuntadores a los registros del archivo de datos, creándose así un nivel de indirección adicional.

Los nodos hoja del árbol B+ suelen estar enlazados para ofrecer un acceso ordenado a los registros según el campo de búsqueda. Estos nodos hoja son similares al primer nivel (base) de un índice. Los nodos internos del árbol B+ corresponden a los demás niveles del

índice. Algunos valores del campo de búsqueda de los nodos hoja se repiten en los nodos internos del árbol B+ con el fin de guiar la búsqueda. La estructura de los *nodos internos* de un árbol B+ de orden p (Fig. 5.11(a)) se define como sigue:

1. Todo nodo interno tiene la forma

   $$<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$$

   donde $q \leq p$ y cada P_i es un **apuntador de árbol**.
2. Dentro de cada nodo interno, $K_1 < K_2 < \cdots < K_{q-1}$.
3. Para todos los valores X del campo de búsqueda en el subárbol al que apunta P_i, tenemos $K_{i-1} < X \leq K_i$, para $1 < i < q$, $X \leq K_i$, para $i = 1$, y $K_{i-1} < X$, para $i = q$ (véase la Fig. 5.11(a)).†
4. Cada nodo interno tiene cuando más p apuntadores de árbol.
5. Cada nodo interno, con excepción de la raíz, tiene por lo menos $\lceil (p/2) \rceil$ apuntadores de árbol. El nodo raíz tiene por lo menos dos apuntadores de árbol si es un nodo interno.
6. Un nodo interno con q apuntadores, donde $q \leq p$, tiene $q - 1$ valores del campo de búsqueda.

La estructura de los *nodos hoja* de un árbol B+ de orden p (Fig. 5.11(b)) es como sigue:

1. Todo nodo hoja tiene la forma

   $$< <K_1, Pr_1>, <K_2, Pr_2>, ..., <K_{q-1}, Pr_{q-1}>, P_{siguiente} >$$

   donde $q \leq p$, cada Pr_i es un apuntador de datos y $P_{siguiente}$ apunta al siguiente *nodo hoja* del árbol B+.
2. Dentro de cada nodo hoja, $K_1 < K_2 < \cdots < K_{q-1}$, donde $q \leq p$.

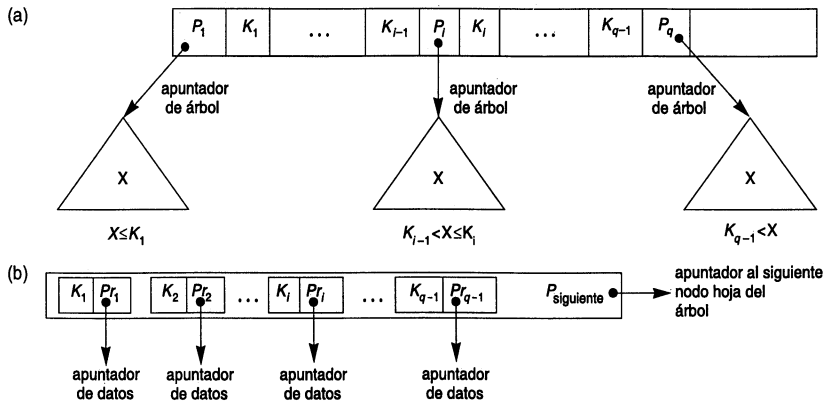


Figura 5.11 Nodos de un árbol B+. (a) Nodo interno de un árbol B+ con $q - 1$ valores de búsqueda. (b) Nodo hoja de un árbol B+.

†Nuestra definición se basa en la de Knuth (1973). Se puede definir un árbol B+ de otra manera intercambiando los símbolos < y ≤ ($K_{i-1} \leq X < K_i$; $X < K_i$; $K_{q-1} \leq X$), pero los principios siguen siendo los mismos.

3. Cada Pr_i es un **apuntador de datos** que apunta al registro cuyo valor de clave de búsqueda es K_i, o a un bloque de archivo que contiene dicho registro (o a un bloque de apuntadores que apuntan a registros cuyo valor del campo de búsqueda es K_i, si el campo de búsqueda no es clave).
4. Cada nodo hoja tiene por lo menos $\lfloor (p/2) \rfloor$ valores.
5. Todos los nodos hoja están en el mismo nivel.

En el caso de un árbol B+ construido según un campo clave, los apuntadores de los nodos internos son *apuntadores de árbol* a bloques que son nodos del árbol, en tanto que los apuntadores de los nodos hoja son *apuntadores de datos* a los registros o bloques del archivo de datos, con la excepción del apuntador $P_{siguiente}$, que es un apuntador de árbol al siguiente nodo hoja. Si partimos del nodo hoja del extremo izquierdo, podremos recorrer los nodos hoja como si fueran una lista enlazada mediante los apuntadores $P_{siguiente}$. Esto hace posible un acceso ordenado a los registros de datos según el campo de indización. También podemos incluir un apuntador $P_{anterior}$. En el caso de un árbol B+ según un campo no clave se requiere un nivel adicional de indirección similar al que se muestra en la figura 5.5, de modo que los apuntadores *Pr* sean apuntadores a bloques que contengan un conjunto de apuntadores a los registros reales del archivo de datos, como se explicó en la opción 3 de la sección 5.1.3.

Como las entradas en los *nodos internos* de un árbol B+ contienen valores de búsqueda y apuntadores de árbol, pero no apuntadores de datos, es posible empaquetar más entradas en un nodo interno de un árbol B+ que en un nodo similar de un árbol B. Por tanto, si el tamaño de bloque (nodo) es el mismo, el orden p será mayor para el árbol B+ que para el árbol B, como se ilustra en el ejemplo 6. Esto puede reducir el número de niveles del árbol B+, mejorándose así el tiempo de búsqueda. Como las estructuras de los nodos internos y los nodos hoja de los árboles B+ son diferentes, el orden p puede ser. Denotaremos con p el orden de los *nodos internos* y con p_{hoja} el orden de los *nodos hoja*, el cual definimos como el número máximo de apuntadores de datos en un nodo hoja.

EJEMPLO 6: Para calcular el orden p de un árbol B+, supongamos que el campo de clave de búsqueda tiene $V = 9$ bytes de largo, que el tamaño de bloque es $B = 512$ bytes, que un apuntador a registro tiene $P_r = 7$ bytes y que un apuntador a bloque tiene $P = 6$ bytes, como en el ejemplo 4. Un nodo interno del árbol B+ puede tener hasta p apuntadores de árbol y $p - 1$ valores del campo de búsqueda. Éstos deben caber en un solo bloque, de modo que tenemos:

$$(p * P) + ((p - 1) * V) \leq B$$
$$(p * 6) + ((p - 1) * 9) \leq 512$$
$$(15 * p) \leq 521$$

Podemos elegir que p sea el valor más grande que satisfaga la desigualdad anterior, lo que nos da $p = 34$. Esto es mayor que el valor de 23 del árbol B, lo que resulta en un abanico más grande y un mayor número de entradas en cada nodo interno de un árbol B+, en comparación con el árbol B correspondiente. Los nodos hoja del árbol B+ tendrán el mismo número de valores y de apuntadores, sólo que en este caso los apuntadores son apuntadores de datos y un apuntador al siguiente. De tal modo, el orden p_{hoja} de los nodos hoja se puede calcular así:

$$(p_{hoja} * (P_r + V)) + P \leq B$$

$$(p_{hoja} * (7 + 9)) + 6 \leq 512$$
$$(16 * p_{hoja}) \leq 506$$

De ello se sigue que cada nodo hoja puede contener hasta $p_{hoja} = 31$ combinaciones de valor de clave y apuntador de datos, suponiendo que los apuntadores de datos son apuntadores a registros. ■

Al igual que con los árboles B, quizá necesitemos información adicional —para implementar los algoritmos de inserción y eliminación— en cada nodo. Esta información puede incluir el tipo de nodo (interno u hoja), el número de entradas q que hay actualmente en el nodo, y apuntadores a los nodos padre y hermanos. Entonces, antes de efectuar los cálculos anteriores de p y p_{hoja}, tendremos que reducir el tamaño del bloque restándole los bytes necesarios para guardar dicha información. El siguiente ejemplo ilustra la manera de calcular el número de entradas de un árbol B+.

EJEMPLO 7: Supongamos que construimos un árbol B+ según el campo del ejemplo 6. Para calcular el número aproximado de entradas del árbol B+, suponemos que cada nodo está ocupado al 69% de su capacidad. En promedio, cada nodo interno tendrá 34 * 0.69 o aproximadamente 23 apuntadores, y por tanto 22 valores. Cada nodo hoja, en promedio, contendrá 0.69 * p_{hoja} = 0.69 * 31 o aproximadamente 21 apuntadores a registros de datos. Un árbol B+ tendrá los siguientes números medios de entradas en cada nivel:

Raíz:	1 nodo	22 entradas	23 apuntadores
Nivel 1:	23 nodos	506 entradas	529 apuntadores
Nivel 2:	529 nodos	11 638 entradas	12 167 apuntadores
Nivel de hoja:	12 167 nodos	255 507 entradas	

Con los tamaños de bloque, apuntador y campo de búsqueda antes dados, un árbol B+ de tres niveles contiene hasta 255 507 apuntadores a registros, en promedio. Compárese esto con las 65 535 entradas del árbol B correspondiente del ejemplo 5. ■

Búsqueda, inserción y eliminación con árboles B+. El algoritmo 5.2 bosqueja el procedimiento en el que el árbol B+ funciona como estructura de acceso para buscar un registro. El algoritmo 5.3 ilustra el procedimiento para insertar un registro en un archivo con una estructura de acceso de árbol B+. Para estos algoritmos se supone que existe un campo clave de búsqueda, y habrá que modificarlos de manera apropiada si el árbol B+ se construye según un campo no clave. Ahora ilustraremos la inserción y la eliminación con un ejemplo.

ALGORITMO 5.2 Búsqueda de un registro con valor de campo clave de búsqueda K, empleando un árbol B+.

```
n ← bloque que contiene el nodo raíz del árbol B+;
leer bloque n;
mientras (n no sea nodo hoja del árbol B+) hacer
  comenzar
  q ← número de apuntadores de árbol del nodo n;
  si K ≤ n.K1 (* n.Ki se refiere al i-ésimo valor del campo de búsqueda en el
  nodo n*)
      entonces n ← n.P1 (* n.Pi se refiere al i-ésimo apuntador de árbol en el
      nodo n*)
```



```
      si no si K > n.Kq-1
          entonces n ← n.Pq
          si no comenzar
              buscar en el nodo n una entrada i tal que n.Ki-1 < K ≤ n.Ki;
              n ← n.Pi
              terminar;
  leer bloque n
  terminar;
buscar en el bloque n la entrada (Ki, Pri) con K = Ki; (* buscar en el nodo hoja *)
si se encuentra
  entonces leer el bloque del archivo de datos con dirección Pri y obtener el
  registro
  si no el registro con valor de campo de búsqueda K no está en el archivo de datos;
```

ALGORITMO 5.3 Inserción de un registro con valor del campo clave de búsqueda K en un árbol B+ de orden p.

```
n ← bloque que contiene el nodo raíz del árbol B+;
leer bloque n; ajustar la pila S a vacía;
mientras (n no sea un nodo hoja del árbol B+) hacer
  comenzar
  meter la dirección de n en la pila S;
      (* la pila S contiene los nodos padre que se requerirán en caso de haber
      división *)
  q ← número de apuntadores de árbol en el nodo n;
  si K ≤ n.K1 (* n.Ki se refiere al i-ésimo valor del campo de búsqueda en el nodo n *)
      entonces n ← n.P1 (* n.Pi se refiere al i-ésimo apuntador de árbol en el nodo n *)
      si no si K > n.Kq-1
          entonces n ← n.Pq
          si no comenzar
              buscar en el nodo n una entrada i tal que n.Ki-1 < K ≤ n.Ki;
              n ← n.Pi
              terminar;
  leer bloque n
  terminar;
buscar en el bloque n la entrada (Ki, Pri) con K = Ki; (* buscar en el nodo hoja n *)
si se encuentra
  entonces el registro ya está en el archivo: imposible insertar
  si no (* insertar en el árbol B+ una entrada que apunte al registro *)
  comenzar
  crear la entrada (K, Pr) donde Pr apunta al nuevo registro;
  si el nodo hoja n no está lleno
  entonces insertar la entrada (K, Pr) en la posición correcta dentro del nodo hoja n
  si no
  comenzar (* el nodo hoja n está lleno con phoja apuntadores a registros: se divide *)
  copiar n en temp (* temp es un nodo hoja grande en el que cabe una entrada
  adicional *);
```

```
    insertar la entrada (K, Pr) en temp en la posición correcta;
        (* temp ahora contiene p_hoja + 1 entradas de la forma (K_i, Pr_i) *)
    nuevo ← un nuevo nodo hoja vacío para el árbol;
    j ← ⌈((p_hoja + 1)/2)⌉;
    n ← primeras j entradas de temp (hasta la entrada (K_j, Pr_j)); n.P_siguiente ← nuevo;
    nuevo ← entradas restantes de temp; K ← K_j;
(* ahora debemos pasar (K, nuevo) al nodo interno padre; sin embargo, si el
padre está lleno, la división puede propagarse *)
    terminamos ← falso;
    repetir
    si la pila S está vacía
        entonces (* no hay nodo padre; se crea un nuevo nodo raíz para el árbol *)
        comenzar
        raíz ←un nuevo nodo interno vacío para el árbol;
        raíz ← <n, K, nuevo>; terminamos ← verdadero;
        terminar
    si no
        comenzar
        n ← sacar de la pila S;
        si el nodo interno n no está lleno
            entonces
            comenzar (* el nodo padre no está lleno: no habrá división *)
            insertar (K, nuevo) en la posición correcta dentro del nodo interno n;
            terminamos ← verdadero
            terminar
            si no
            comenzar (* el nodo interno n está lleno con p apuntadores de árbol:
            se dividirá *)
            copiar n en temp (* temp es un nodo interno grande *);
            insertar (K, nuevo) en temp en la posición correcta;
                (* temp tiene ahora p + 1 apuntadores de árbol *)
            nuevo ← un nuevo nodo interno vacío para el árbol;
            j ← ⌊((p + 1)/2)⌋;
            n ← entradas hasta el apuntador de árbol P_j de temp;
                (* n contiene <P_1, K_1, P_2, K_2, ..., P_{j-1}, K_{j-1}, P_j> *)
            nuevo ← entradas a partir del apuntador de árbol P_{j+1} de temp;
                (* nuevo contiene <P_{j+1}, K_{j+1}, ..., K_{p-1}, P_p, K_p, P_{p+1}> *)
            K ← K_j
        (* ahora debemos pasar (K, nuevo) al nodo interno padre *)
            terminar
        terminar
    hasta terminamos
    terminar;
terminar;
```

La figura 5.12 ilustra la inserción de registros en un árbol B⁺ de orden $p = 3$ y $p_{hoja} = 2$. Primero, observamos que la raíz es el único nodo en el árbol, así que también es un nodo hoja.

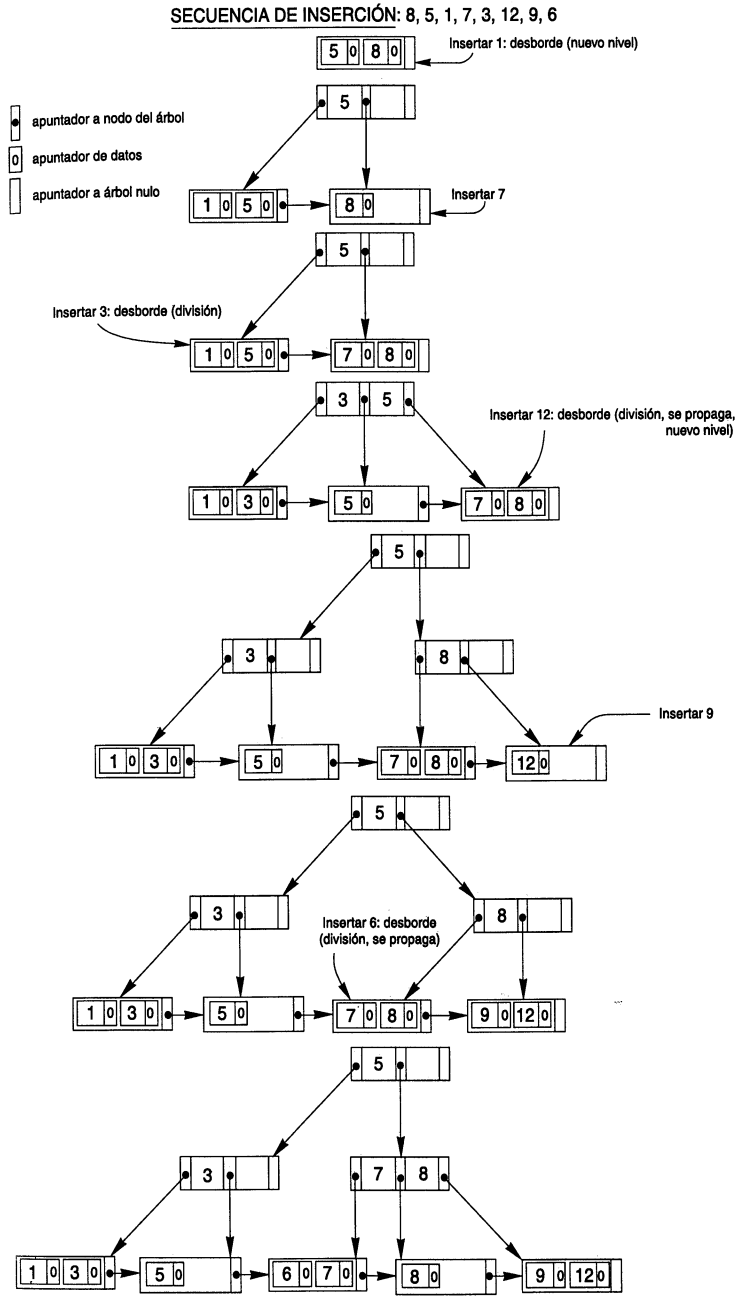


Figura 5.12 Ejemplo de inserción en un árbol B⁺.

Tan pronto como se crea más de un nivel, el árbol se divide en nodos internos y nodos hoja. Observe que *todo valor debe existir en el nivel de hoja*, porque todos los apuntadores de datos están en ese nivel. Sin embargo, sólo existen algunos valores en los nodos internos para guiar la búsqueda. Advierta también que todos los valores que aparecen en un nodo interno aparecen también como *el valor del extremo derecho* en el subárbol al que apunta el apuntador de árbol que está a la izquierda del valor.

Cuando se llena un *nodo hoja* y se inserta en él una nueva entrada, el nodo **se desborda** y hay que dividirlo. Las primeras $j = \lceil ((p_{hoja} + 1)/2) \rceil$ entradas en el nodo original se mantienen ahí, y las demás se pasan a un nuevo nodo hoja. El j-ésimo valor de búsqueda se repite en el nodo interno padre, y se crea en él un apuntador adicional al nuevo nodo. Éstos deben insertarse en el nodo padre en su secuencia correcta. Si el nodo interno padre está lleno, el nuevo valor hará que se desborde también, y tendrá que dividirse. Las entradas en el nodo interno hasta P_j —el j-ésimo apuntador de árbol después de insertar el nuevo valor y el nuevo apuntador, donde $j = \lfloor ((p + 1)/2) \rfloor$— se conservan, en tanto que el j-ésimo valor de búsqueda se *pasa* al padre, sin replicación. Un nuevo nodo interno contendrá las entradas desde P_{j+1} hasta el final de las entradas del nodo (véase el algoritmo 5.3). Esta división puede propagarse hasta arriba para crear un nuevo nodo raíz y, por tanto, un nuevo nivel del árbol B+.

La figura 5.13 ilustra la eliminación en un árbol B+. Cuando se elimina una entrada, siempre se hace del nivel de hoja. Si aparece en un nodo interno, también habrá que quitarla de ahí. En este caso, el valor que está a su izquierda en el nodo hoja debe reemplazarlo en el nodo interno, porque ahora es la entrada del extremo derecho del subárbol. La eliminación puede causar **insuficiencia** si reduce el número de entradas del nodo hoja por debajo del mínimo requerido. En este caso trataremos de encontrar un nodo hoja **hermano** —uno que esté inmediatamente a la derecha o a la izquierda del nodo con insuficiencia— y de **redistribuir** las entradas entre el nodo y su hermano de modo que ambos estén ocupados por lo menos hasta la mitad; si esto no es posible, el nodo se fusionará con sus hermanos, reduciéndose así el número de nodos hoja. Un método común consiste en tratar de redistribuir las entradas con el hermano izquierdo; si esto no es posible, se intenta redistribuirlas con el hermano derecho. Si tampoco esto es factible, los tres nodos se fusionan para formar dos nodos hoja. En un caso así, es posible que la insuficiencia se propague a los nodos **internos**, al requerirse un apuntador de árbol y un valor de búsqueda menos. Esto puede propagarse y reducir el número de niveles del árbol.

Observe que la implementación de los algoritmos para insertar y eliminar puede requerir apuntadores al padre y a los hermanos en todos los nodos, o el empleo de una pila como en el algoritmo 5.3. Cada nodo deberá incluir también el número de entradas que contiene y su tipo (hoja o interno). Otra alternativa es implementar la inserción y la eliminación como procedimientos recursivos.

Variaciones de los árboles B y B+. Para concluir esta sección, hagamos una breve mención de algunas variaciones de los árboles B y B+. En algunos casos la restricción 5 de los árboles B (o B+), que obliga a todos los nodos a estar ocupados por lo menos hasta la mitad, se puede modificar de modo que exija que todos los nodos estén ocupados por lo menos hasta las dos terceras partes de su capacidad. A este tipo de árboles B se les ha llamado **árboles B***. En general, algunos sistemas permiten que el usuario elija un **factor de llenado** de entre 0.5 y 1.0, en donde la segunda cifra significa que los nodos del árbol B (o del índice) deben estar completamente llenos. Por añadidura, con algunos sistemas el usuario puede

especificar dos factores de llenado para los árboles B+: uno para el nivel de hoja y otro para los nodos internos del árbol. Al construirse inicialmente el índice, todos los nodos se ocupan hasta alcanzar aproximadamente los factores de llenado especificados. En fechas recientes, algunos investigadores han sugerido que el requerimiento de que un nodo esté lleno hasta la mitad sea menos riguroso, y se permita que llegue a estar completamente vacío antes de efectuarse una fusión, a fin de simplificar el algoritmo de eliminación. Hay estudios que indican que esto no desperdicia demasiado espacio si las inserciones y eliminaciones se distribuyen en forma aleatoria.

5.4 Otros tipos de índices*

5.4.1 *Empleo de la dispersión y de otras estructuras de datos como índices*

También es posible crear estructuras de acceso similares a índices, basadas en la *dispersión*. Las entradas de índice <*K*, *Pr*> (o <*K*, *P*>) se pueden organizar en forma de archivo de dispersión dinámicamente expansible mediante alguna de las técnicas descritas en la sección 4.8.3. Para buscar una entrada, aplicaremos el algoritmo de dispersión a *K*; una vez localizada la entrada, el apuntador *Pr* (o *P*) nos servirá para encontrar el registro correspondiente en el archivo de datos. También podemos usar otras estructuras de búsqueda como índices.

5.4.2 *Índices lógicos vs. físicos*

Hasta ahora hemos supuesto que las entradas de índice <*K*, *Pr*> (o <*K*, *P*>) siempre incluyen un apuntador físico *Pr* (o *P*) que especifica la dirección del registro físico en el disco en forma de un número de bloque y un desplazamiento. Esto se conoce como **índice físico**, y tiene la desventaja de que el apuntador debe modificarse si el registro se pasa a otro lugar del disco. Por ejemplo, supongamos que la organización primaria de un archivo se basa en una dispersión lineal o extensible; entonces, cada vez que se divida una cubeta, algunos registros se asignarán a cubetas nuevas, y por tanto tendrán nuevas direcciones físicas. Si el archivo tiene un índice secundario, será necesario encontrar y actualizar los apuntadores a esos registros, tarea nada sencilla.

A fin de remediar esta situación, podemos usar una estructura llamada **índice lógico**, cuyas entradas tienen la forma <K, K_p>. Cada entrada tiene un valor K para el campo de indización secundaria apareado con el valor K_p del campo empleado para la organización primaria del archivo. Si un programa busca el valor K en el índice secundario, podrá localizar el valor correspondiente de K_p y utilizarlo para tener acceso al registro valiéndose de la organización primaria del archivo. Los índices lógicos se utilizan cuando se espera que las direcciones físicas de los registros cambien con frecuencia. El costo es la búsqueda adicional que se basa en la organización primaria del archivo.

5.4.3 *Análisis*

En muchos sistemas, el índice no es parte integral del archivo de datos, sino que puede crearse y desecharse dinámicamente; es por ello que se acostumbra llamarlo estructura de acceso.

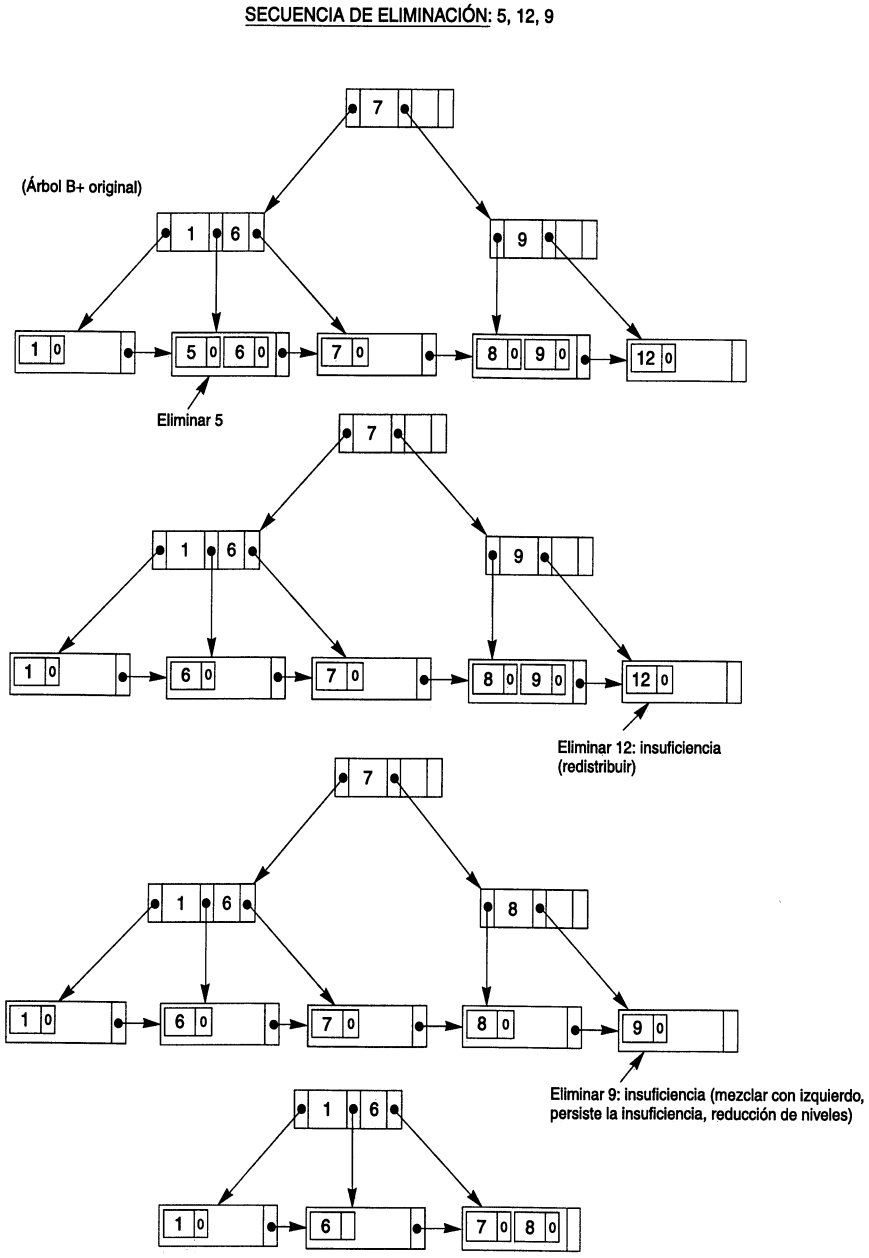


Figura 5.13 Ejemplo de eliminación en un árbol B^+.

Siempre que esperemos requerir acceso frecuente a un archivo según una condición de búsqueda que implique un campo en particular, podemos solicitar al SGBD la creación de un índice basado en ese campo. Por lo regular, se creará un índice secundario con el fin de evitar la ordenación física de los registros del archivo de datos.

La principal ventaja de los índices secundarios es que —por lo menos en teoría— se pueden crear con *casi cualquier organización de registros*. Por tanto, el índice secundario puede servir para complementar otros métodos de acceso primarios, como el ordenamiento o la dispersión, e incluso puede usarse con archivos mixtos. Para crear un índice secundario de árbol B^+ con base en algún campo de un archivo, deberemos examinar todos los registros del archivo y crear las entradas a nivel de hoja del árbol. A continuación, dichas entradas se ordenarán y llenarán de acuerdo con el factor de llenado prescrito, creándose simultáneamente los demás niveles del índice. Resulta más costoso y mucho más difícil crear dinámicamente índices primarios y de agrupamiento, porque los registros del archivo de datos deben estar ordenados físicamente en el disco según el campo de indización. Con todo, algunos sistemas permiten a los usuarios crear dinámicamente estos índices en sus archivos.

A menudo se utilizan los índices para imponer una *restricción de clave* al campo de índice de un archivo. Cuando se busca en el índice el lugar donde se insertará un registro nuevo, resulta sencillo verificar al mismo tiempo si algún otro registro del archivo —y por tanto del árbol— tiene el mismo valor para el campo de indización. Si es así, la inserción podrá rechazarse.

A un archivo que tiene un índice secundario para cada uno de sus campos suele llamársele **archivo totalmente invertido.** Como todos los índices son secundarios, los registros nuevos se insertan al final del archivo; por tanto, el archivo de datos en sí es un archivo no ordenado (de montículo). Los índices casi siempre se implementan en forma de árboles B^+, por lo que se actualizan dinámicamente para reflejar la inserción o eliminación de registros.

Otra organización común es la de un archivo ordenado con un índice primario de múltiples niveles basado en su campo clave de ordenamiento. Esta organización suele recibir el nombre de **archivo secuencial indizado** y se le utiliza comúnmente en el procesamiento de datos de negocios. La inserción se realiza con la ayuda de algún tipo de archivo de desborde que se fusiona periódicamente con el archivo de datos. El índice se vuelve a crear durante la reorganización del archivo. El **método de acceso secuencial indizado** (ISAM: *indexed sequential access method*) de IBM incorpora un índice de dos niveles que está íntimamente relacionado con el disco. El primer nivel es un índice de cilindros, que tiene el valor de clave de un registro ancla por cada cilindro del paquete de discos y un apuntador al índice de pistas del cilindro; éste índice tiene el valor de clave de un registro ancla por cada pista del cilindro y un apuntador a la pista. Así, el registro o bloque deseado puede buscarse secuencialmente en la pista. Otro método de IBM, el **método de acceso de almacenamiento virtual** (VSAM: *virtual storage access method*) se parece un poco a la estructura de acceso de árbol B^+.

5.5 Resumen

En este capítulo presentamos organizaciones de archivo en las que intervienen estructuras de acceso adicionales, los índices, con los que se mejora la eficiencia en la obtención de registros de un archivo de datos. Dichas estructuras de acceso pueden usarse *junto con* las organizaciones primarias de archivos que vimos en el capítulo 4, las cuales sirven para organizar los registros mismos del archivo en el disco.

Primero analizamos tres tipos de índices ordenados de un solo nivel: primarios, secundarios y de agrupamiento. Cada índice se basa en un campo del archivo. Los índices primarios y de agrupamiento se construyen según el campo de ordenamiento físico del archivo, en tanto que los índices secundarios se basan en campos que no son de ordenamiento. El campo de un índice primario debe ser además una clave del archivo, cosa que no sucede con un índice de agrupamiento. Los índices de un solo nivel son archivos ordenados y se examinan con una búsqueda binaria. Mostramos cómo se construyen índices de múltiples niveles para mejorar la eficiencia de las búsquedas.

Después explicamos la implementación de los índices de múltiples niveles en forma de árboles B y B^+, que son estructuras dinámicas que permiten al índice expandirse y contraerse dinámicamente. Los nodos (bloques) de estas estructuras de índice se mantienen ocupados entre el 50% y el 100% de su capacidad gracias a sus algoritmos de inserción y eliminación. Después de cierto tiempo, los nodos se estabilizan en un grado de ocupación promedio del 69%, lo que deja espacio para hacer inserciones sin tener que reorganizar el índice con mucha frecuencia. En general, los árboles B^+ pueden contener más entradas en sus nodos internos que los árboles B, por lo que es posible que un árbol B^+ tenga menos niveles o incluya más entradas que el árbol B correspondiente.

En la sección 5.4 explicamos la forma de construir un índice con base en estructuras de datos de dispersión. A continuación, presentamos el concepto de índice lógico, comparándolo con los índices físicos antes descritos. Por último, vimos la forma de utilizar combinaciones de las organizaciones anteriores; por ejemplo, los índices secundarios suelen usarse con archivos mixtos, así como con archivos ordenados y no ordenados. También se pueden crear índices secundarios para archivos de dispersión y de dispersión dinámica.

Preguntas de repaso

5.1. Defina los siguientes términos: *campo de indización, campo de clave primaria, campo de agrupamiento, campo de clave secundaria, ancla de bloque, índice denso, índice no denso.*

5.2. ¿Qué diferencias hay entre los índices primarios, secundarios y de agrupamiento? ¿De qué manera afectan dichas diferencias las formas de implementar esos índices? ¿Cuáles de esos índices son densos, y cuáles no?

5.3. ¿Por qué no podemos tener más de un índice primario o de agrupamiento de un archivo, pero sí varios índices secundarios?

5.4. ¿Cómo mejora la eficiencia de las búsquedas en un archivo de índice con la indización de múltiples niveles?

5.5. ¿Qué es el orden *p* de un árbol B? Describa la estructura de los nodos de los árboles B.

5.6. ¿Qué es el orden *p* de un árbol B^+? Describa la estructura de los nodos internos y de los nodos hoja de los árboles B^+.

5.7. ¿Qué diferencia hay entre un árbol B y un árbol B^+? ¿Por qué suelen preferirse los árboles B^+ como estructuras de acceso a los archivos de datos?

5.8. ¿Qué es un archivo totalmente invertido? ¿Qué es un archivo secuencial indizado?

5.9. ¿Cómo podemos usar la dispersión para construir un índice? ¿Qué diferencia hay entre un índice lógico y uno físico?

Ejercicios

5.10. Considere un disco con bloques de $B = 512$ bytes. Un apuntador de bloque tiene $P = 6$ bytes de largo, y un apuntador a registro tiene $P_R = 7$ bytes de largo. Un archivo tiene 30 000 registros EMPLEADO de *longitud fija*. Cada registro tiene los siguientes campos: NOMBRE (30 bytes), NSS (9 bytes), CÓDIGODEPTO (9 bytes), DIRECCIÓN (40 bytes), TELÉFONO (9 bytes), FECHANAC (8 bytes), SEXO (1 byte), CÓDIGOPUESTO (4 bytes), SALARIO (4 bytes, número real). Se utiliza un byte adicional como marcador de eliminación.

a. Calcule el tamaño de registro *R* en bytes.

b. Calcule el factor de bloques, *fbl*, y el número de bloques de archivo, *b*, suponiendo una organización no extendida.

c. Suponga que el archivo está *ordenado* según el campo clave NSS y que deseamos construir un *índice primario* basado en NSS. Calcule (i) el factor de bloques del índice, fbl_i (que también es el abanico del índice, *fo*); (ii) el número de entradas de primer nivel del índice y el número de bloques de primer nivel del índice; (iii) el número de niveles que necesitaremos si creamos un índice de múltiples niveles; (iv) el número total de bloques que requiere el índice de múltiples niveles, y (v) el número de accesos a bloques necesarios para buscar un registro del archivo y obtenerlo —dado su valor de NSS— empleando el índice primario.

d. Suponga que el archivo *no está ordenado* según el campo clave NSS y que deseamos construir un *índice secundario* basado en dicho campo. Repita el ejercicio anterior (parte c) para el índice secundario y haga una comparación con el índice primario.

e. Suponga que el archivo *no está ordenado* según el campo no clave CÓDIGODEPTO y que deseamos construir un *índice secundario* basado en dicho campo, eligiendo la opción 3 de la sección 5.1.3, con un nivel adicional de indirección que almacene apuntadores a registros. Suponga que hay 1000 valores distintos de CÓDIGODEPTO y que los registros de EMPLEADO están distribuidos de manera uniforme entre esos valores. Calcule (i) el factor de bloques del índice, fbl_i (que también es el abanico del índice, *fo*); (ii) el número de bloques necesarios para el nivel de indirección que almacena apuntadores a registros; (iii) el número de entradas de primer nivel del índice y el número de bloques de primer nivel del índice; (iv) el número de niveles que necesitaremos si creamos un índice de múltiples niveles; (v) el número total de bloques que requieren el índice de múltiples niveles y el nivel adicional de indirección, y (vi) el número aproximado de accesos a bloques necesarios para buscar y leer *todos* los registros del archivo que tienen un determinado valor de CÓDIGODEPTO, empleando el índice.

f. Suponga que el archivo está *ordenado* según el campo no clave CÓDIGODEPTO y que deseamos construir un *índice de agrupamiento* basado en dicho campo y usando anclas de bloque (cada nuevo valor de CÓDIGODEPTO comienza al principio de un nuevo bloque). Suponga que hay 1000 valores distintos de CÓDIGODEPTO y que los registros EMPLEADO están distribuidos uniformemente entre esos valores. Calcule (i) el factor de bloques del índice, fbl_i (que también es el abanico del índice, *fo*); (ii) el número de entradas de primer nivel del índice y el número de bloques de primer nivel del índice; (iii) el número de niveles que necesitaremos si creamos un índice de múltiples niveles; (iv) el número total de bloques que requiere el índice de

múltiples niveles, y (v) el número de accesos a bloques necesarios para buscar y leer todos los registros del archivo que tienen un determinado valor de CÓDIGODEPTO, empleando el índice de agrupamiento (suponga que si un grupo abarca varios bloques éstos son contiguos o están enlazados mediante apuntadores).

g. Suponga que el archivo *no* está ordenado según el campo clave NSS y que deseamos construir una estructura de acceso (índice) de árbol B^+ basada en NSS. Calcule (i) los órdenes p y p_{hoja} del árbol B^+; (ii) el número de bloques a nivel de hoja requeridos si los bloques están ocupados aproximadamente al 69% de su capacidad (redondeado hacia arriba por comodidad); (iii) el número de niveles requeridos si los nodos internos también están ocupados al 69% (redondeado hacia arriba por comodidad); (iv) el número total de bloques que ocupa el árbol B^+, y (v) el número de accesos a bloques necesarios para buscar y leer un registro del archivo —dado su valor de NSS— empleando el árbol B^+.

h. Repita la parte g, pero con un árbol B en vez de un árbol B^+. Compare los resultados de los dos casos.

5.11. Un archivo COMPONENTES con NúmComp como campo clave contiene registros con los siguientes valores de NúmComp: 23, 65, 37, 60, 46, 92, 48, 71, 56, 59, 18, 21, 10, 74, 78, 15, 16, 20, 24, 28, 39, 43, 47, 50, 69, 75, 8, 49, 33, 38; suponga que los valores de campo están insertados en este orden en un árbol B^+ de orden $p = 4$ y $p_{hoja} = 3$. Indique la forma en que se expandirá el árbol y muestre el aspecto final que tendrá.

5.12. Repita el ejercicio 5.11, pero utilice un árbol B de orden $p = 4$ en vez de un árbol B^+.

5.13. Suponga que se eliminan, en el orden dado, ciertos valores del campo de búsqueda del árbol B^+ del ejercicio 5.11. Indique cómo se contraerá el árbol y muestre el árbol final. Los valores eliminados son 65, 75, 43, 18, 20, 92, 59, 37.

5.14. Repita el ejercicio 5.13, pero con el árbol B del ejercicio 5.12.

5.15. El algoritmo 5.1 bosqueja el procedimiento para efectuar una búsqueda en un índice primario no denso de múltiples niveles con el fin de obtener un registro del archivo. Adapte el algoritmo a cada uno de los siguientes casos:

a. Un índice secundario de múltiples niveles según un campo no clave que no es el campo de ordenamiento de un archivo. Suponga que se usa la opción 3 de la sección 5.1.3, donde un nivel adicional de indirección almacena apuntadores a los registros individuales que tienen el valor del campo de indización correspondiente.

b. Un índice secundario de múltiples niveles según un campo clave que no es el campo de ordenamiento de un archivo.

c. Un índice de agrupamiento de múltiples niveles según un campo no clave que sí es el campo de ordenamiento de un archivo.

5.16. Suponga que un archivo tiene varios índices secundarios según campos no clave, implementados empleando la opción 3 de la sección 5.1.3; por ejemplo, podríamos tener índices secundarios según los campos CÓDIGODEPTO, CÓDIGOPUESTO y SALARIO del archivo EMPLEADO del ejercicio 5.10. Describa una manera eficiente de buscar y obtener los registros que satisfagan una condición de selección compleja expresada en términos de esos campos, como (CÓDIGODEPTO = 5 Y CÓDIGOPUESTO = 12 Y SALARIO > 50 000), empleando los apuntadores a registros del nivel de indirección.

5.17. Adapte los algoritmos 5.2 y 5.3, que bosquejan procedimientos de búsqueda e inserción en un árbol B^+, a un árbol B.

5.18. Es posible modificar el algoritmo de inserción en un árbol B^+ a fin de postergar el caso en el que se producirá un nuevo nivel verificando si es factible una *redistribución* de los valores entre los nodos hoja. La figura 5.14 ilustra la forma en que esto podría hacerse con nuestro ejemplo de la figura 5.12; en vez de dividir el nodo hoja del extremo izquierdo cuando se inserta el 12, efectuamos una *redistribución a la izquierda* pasando el 7 al nodo hoja que está a su izquierda (si hay espacio en este nodo). La figura 5.14 muestra el aspecto que tendría el árbol si se considerara la redistribución. También es posible considerar una *redistribución a la derecha*. Trate de modificar el algoritmo de inserción en un árbol B^+ de modo que tenga en cuenta la redistribución.

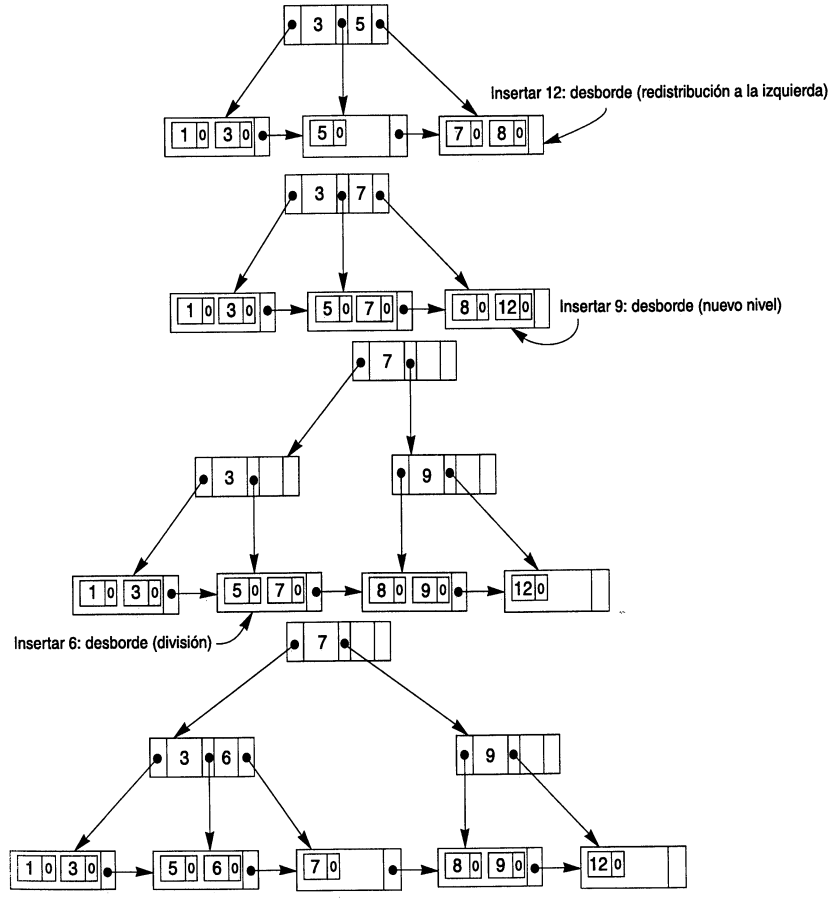


Figura 5.14 Inserción en un árbol B^+ con redistribución a la izquierda.

5.19. Bosqueje un algoritmo para eliminar entradas de un árbol B^+.

5.20. Repita el ejercicio 5.19 con un árbol B.

Bibliografía selecta

Nievergelt (1974) analiza el empleo de árboles de búsqueda binaria para la organización de archivos. Bayer y McCreight (1972) define los árboles B, y Comer (1979) es un estudio de los árboles B, de sus variaciones y de su historia. Knuth (1973) ofrece un análisis detallado de muchas técnicas de búsqueda, incluidos los árboles B y algunas de sus variaciones. Wirth (1972), Salzberg (1988) y Smith y Barnes (1987) presentan algoritmos de búsqueda, inserción y eliminación para árboles B y B^+. Larson (1981) analiza los archivos secuenciales indizados, y Held y Stonebraker (1978) comparan los índices estáticos de múltiples niveles con los índices dinámicos de árbol B. Lehman y Yao (1981) y Srinivasan y Carey (1991) analizan el acceso concurrente a los árboles B. Los libros de Wiederhold (1983), Smith y Barnes (1987) y Salzberg (1988), entre otros, tratan muchas de las técnicas descritas en este capítulo.

Lanka y Mays (1991), Mohan y Narang (1992), Zobel *et al.* (1992) y Faloutsos y Jagadish (1992) analizan nuevas técnicas y aplicaciones de los índices y de los árboles B^+. El rendimiento de diversos algoritmos para árboles B y B^+ se evalúa en Baeza-Yates y Larson (1989) y en Johnson y Shasha (1993). El manejo del almacenamiento intermedio para índices se analiza en Mackert y Lohman (1989) y Chan *et al.* (1992).

CAPÍTULO 6

El modelo de datos relacional y el álgebra relacional

El modelo relacional de los datos fue introducido por Codd (1970). Se basa en una estructura de datos simple y uniforme —la relación— y tiene fundamentos teóricos sólidos. Analizaremos diversos aspectos de este modelo en varios capítulos, ya que hay que cubrir más material conceptual en el caso del modelo relacional que en el de los otros modelos de datos. Además, el modelo relacional se está estableciendo firmemente en el mundo de las aplicaciones de bases de datos, y existen en el mercado muchos paquetes de SGBD relacionales.

En este capítulo nos concentraremos en la descripción de los principios básicos del modelo relacional de los datos; los lenguajes y sistemas relacionales comerciales se tratarán en capítulos subsecuentes. Comenzaremos este capítulo con una definición de los conceptos de modelado y de la notación del modelo relacional en la sección 6.1. En la sección 6.2 identificaremos las restricciones de integridad que ahora se consideran parte importante del modelo relacional. En la sección 6.3 se definen las operaciones de actualización del modelo relacional y se analiza su efecto sobre las restricciones de integridad. En la sección 6.4 se ilustra la forma de declarar las relaciones en un sistema de base de datos. En la sección 6.5 presentaremos un tratamiento detallado del álgebra relacional, que es un conjunto de operaciones para manipular relaciones y especificar consultas. Consideramos el álgebra relacional como una parte integral del modelo de datos relacional. En la sección 6.6 definimos otras operaciones relacionales que son útiles en muchas aplicaciones de bases de datos. En la sección 6.7 daremos ejemplos de la especificación de consultas mediante operaciones relacionales. La sección 6.8 presenta algoritmos para diseñar un esquema de base de datos relacional estableciendo una transformación de un diseño conceptual realizado en el modelo ER (véase el Cap. 3) al modelo relacional. Para concluir, haremos un resumen en la sección 6.9.

Si al lector le interesa una introducción menos detallada a los conceptos relacionales, puede pasar por alto las secciones 6.1.2, 6.5.7 y 6.6. Para la sección 6.8, sobre el diseño de bases de datos relacionales, se supone que el lector conoce el material del capítulo 3, y también puede pasarse por alto.

Toda la parte II de este libro está dedicada al modelo relacional. En el capítulo 7 describiremos el lenguaje de consulta SQL, que se está convirtiendo en la norma para los SGBD relacionales comerciales. Presentaremos otro lenguaje formal para el modelo relacional —el cálculo relacional— en el capítulo 8; este lenguaje proporciona cimientos teóricos para los lenguajes de consulta relacionales comerciales y a partir de él se construyen sistemas relacionales avanzados como las bases de datos deductivas (Cap. 24). También estudiaremos los lenguajes QUEL y QBE en el capítulo 8. Por último, en el capítulo 9 se expondrá un panorama de un SGBD relacional comercial. Los capítulos 12 y 13 de la parte IV del libro presentan las restricciones formales de las dependencias funcionales y multivaluadas y explican cómo se utilizan éstas para desarrollar una teoría, basada en la normalización, para el diseño de bases de datos relacionales.

6.1 Conceptos del modelo relacional

El modelo relacional representa la base de datos como una colección de relaciones. En términos informales, cada relación semeja una tabla o, hasta cierto punto, un archivo simple. Por ejemplo, se considera que la base de datos de archivos que se muestra en la figura 1.2 cae dentro del modelo relacional. Sin embargo, existen diferencias importantes entre las relaciones y los archivos, como habremos de ver.

Si visualizamos una relación como una **tabla** de valores, cada fila de la tabla representa una colección de valores de datos relacionados entre sí. Dichos valores se pueden interpretar como hechos que describen una entidad o un vínculo entre entidades del mundo real. El nombre de la tabla y los nombres de las columnas ayudan a interpretar el significado de los valores que están en cada fila de la tabla. Por ejemplo, la primera tabla de la figura 1.2 se llama ESTUDIANTE porque cada fila representa hechos acerca de una entidad estudiante en particular. Los nombres de las columnas —Nombre, NúmEstudiante, Grado, Carrera— especifican cómo interpretar los valores de datos de cada fila, con base en la columna en la que se encuentra cada valor. Todos los valores de una columna tienen el mismo tipo de datos.

En la terminología del modelo relacional, una fila se denomina *tupla*, una cabecera de columna es un *atributo* y la tabla es una *relación*. El tipo de datos que describe los tipos de valores que pueden aparecer en cada columna se llama *dominio*. A continuación definiremos estos términos —*dominio*, *tupla*, *atributo* y *relación*— con mayor precisión.

6.1.1 *Dominios, tuplas, atributos y relaciones*

Un **dominio** D es un conjunto de valores atómicos. Por **atómico** queremos decir que cada valor del dominio es indivisible en lo tocante al modelo relacional. Un método común de especificación de los dominios consiste en especificar un tipo de datos al cual pertenecen los valores que constituyen el dominio. También resulta útil especificar un nombre para el dominio que ayude a interpretar sus valores. He aquí algunos ejemplos de dominios:

- Números_telefónicos_de_EUA: El conjunto de números telefónicos de 10 dígitos válidos en los Estados Unidos.
- Números_telefónicos_locales: El conjunto de números telefónicos de siete dígitos válidos dentro de un código de área en particular.

- Números_de_seguro_social: El conjunto de números de seguro social válidos formados por nueve dígitos.
- Nombres: El conjunto de nombres de personas.
- Promedios_de_notas: Valores posibles de los promedios de notas calculados; cada uno debe ser un valor entre 0 y 4.
- Edades_de_empleados: Edades posibles de los empleados de una compañía; cada una debe ser un valor entre 16 y 80 años de edad.
- Departamentos_académicos: El conjunto de departamentos académicos, como Ciencias de la computación, Economía y Física, de una universidad.

Las anteriores son definiciones lógicas de dominios. También debe especificarse un **tipo de datos** o **formato** para cada dominio. Por ejemplo, se puede declarar el tipo de datos del dominio Números_telefónicos_de_EUA como una cadena de caracteres de la forma (ddd)ddd-dddd, donde cada d es un dígito numérico (decimal) y los primeros tres dígitos forman un código de área telefónica válido. El tipo de datos de Edades_de_empleados es un número entero entre 16 y 80. En el caso de Departamentos_académicos, el tipo de datos es el conjunto de todas las cadenas de caracteres que representan nombres o códigos válidos de departamentos.

Así pues, un dominio debe tener un nombre, un tipo de datos y un formato. También puede incluirse información adicional para interpretar los valores de un dominio; por ejemplo, un dominio numérico como Pesos_de_personas deberá especificar las unidades de medición: libras o kilogramos. A continuación definiremos el concepto de esquema de relación, que describe la estructura de una relación.

Un **esquema de relación** R, denotado por $R(A_1, A_2, ..., A_n)$, se compone de un nombre de relación, R, y una lista de atributos, $A_1, A_2, ..., A_n$. Cada **atributo** A_i es el nombre de un papel desempeñado por algún dominio D en el esquema R. Se dice que D es el **dominio** de A_i y se denota con $\text{dom}(A_i)$. Un esquema de relación sirve para *describir* una relación; R es el **nombre** de la relación. El **grado de una relación** es el número de atributos, n, de su esquema de relación.

El siguiente es un esquema de relación para una relación de grado 7, que describe estudiantes universitarios:

ESTUDIANTE(Nombre, NSS, TelParticular, Dirección, TelOficina, Edad, Prom)

En este esquema de relación, ESTUDIANTE es el nombre de la relación, la cual tiene siete atributos. Podemos especificar los siguientes dominios para algunos de los atributos de la relación ESTUDIANTE: dom(Nombre) = Nombres; dom(NSS) = Números_de_seguro_social; dom(TelParticular) = Números_telefónicos_locales; dom(TelOficina) = Números_telefónicos_locales; dom(Prom) = Promedios_de_notas.

Una **relación** (o **ejemplar de relación**) r del esquema de relación $R(A_1, A_2, ..., A_n)$, denotado también por $r(R)$, es un conjunto de n-tuplas $r = \{t_1, t_2, ..., t_m\}$. Cada ***n*-tupla** t es una lista ordenada de n valores $t = <v_1, v_2, ..., v_n>$, donde cada valor v_i, $1 \leq i \leq n$, es un elemento de $\text{dom}(A_i)$ o bien un valor **nulo** especial. También se acostumbra usar los términos **intensión** de una relación para el esquema R y **extensión** (o **estado**) de una relación para un ejemplar de relación $r(R)$.

La figura 6.1 muestra un ejemplo de una relación ESTUDIANTE, que corresponde al esquema ESTUDIANTE que acabamos de especificar. Cada tupla de la relación representa una

entidad estudiante en particular. Presentamos la relación en forma de tabla, en la que cada tupla aparece como una fila y cada atributo corresponde a una cabecera de columna que indica un papel o interpretación de los valores de esa columna. Los *valores nulos* representan atributos cuyos valores se desconocen o no existen para algunas tuplas ESTUDIANTE individuales.

La definición anterior de relación puede *expresarse* también como sigue: una relación $r(R)$ es un **subconjunto del producto cartesiano** de los dominios que definen a R:

$$r(R) \subseteq (\text{dom}(A_1) \mathbf{X} \text{dom}(A_2) \mathbf{X} \cdots \mathbf{X} \text{dom}(A_n))$$

El producto cartesiano especifica todas las combinaciones posibles de valores de los dominios implicados. Así pues, si denotamos el número de valores o **cardinalidad** de un dominio D con $|D|$, y suponemos que todos los dominios son finitos, el número total de tuplas del producto cartesiano es

$$|\text{dom}(A_1)| * |\text{dom}(A_2)| * \cdots * |\text{dom}(A_n)|$$

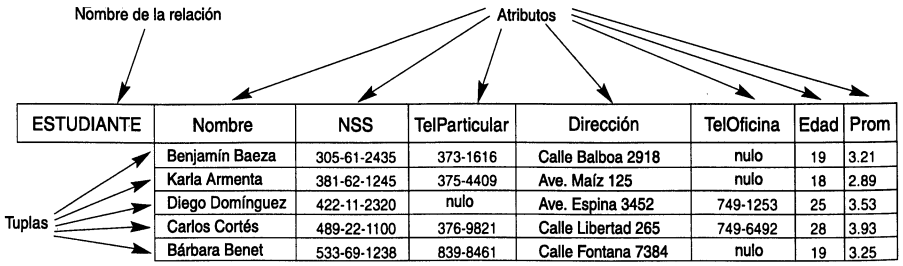
De todas estas posibles combinaciones, un ejemplar de relación en un momento dado —el **estado actual de la relación**— refleja sólo las tuplas válidas que representan un estado específico del mundo real. En general, a medida que cambia el estado del mundo real, cambia la relación, transformándose en otro estado de la relación. Sin embargo, el esquema R es relativamente estático, y *no* cambia con frecuencia; lo hace, por ejemplo, cuando se añade un atributo para representar información nueva que no estaba representada originalmente en la relación.

Es posible que varios atributos *tengan el mismo dominio*. Los atributos indican diferentes **papeles**, o interpretaciones, del dominio. En la relación ESTUDIANTE, por ejemplo, el mismo dominio Números_telefónicos_locales desempeña el papel de TelParticular, refiriéndose al "teléfono particular de un estudiante", y el de TelOficina, refiriéndose al "teléfono de la oficina del estudiante".

6.1.2 *Características de las relaciones*★

La primera definición de relación implica ciertas características que distinguen a una relación de un archivo o de una tabla. A continuación analizaremos algunas de estas características.

Orden de las tuplas en una relación. Una relación se define como un conjunto de tuplas. Matemáticamente, los elementos de un conjunto *no están ordenados*; por tanto, las tuplas de



ESTUDIANTE	Nombre	NSS	TelParticular	Dirección	TelOficina	Edad	Prom
	Benjamín Baeza	305-61-2435	373-1616	Calle Balboa 2918	nulo	19	3.21
	Karla Armenta	381-62-1245	375-4409	Ave. Maíz 125	nulo	18	2.89
	Diego Domínguez	422-11-2320	nulo	Ave. Espina 3452	749-1253	25	3.53
	Carlos Cortés	489-22-1100	376-9821	Calle Libertad 265	749-6492	28	3.93
	Bárbara Benet	533-69-1238	839-8461	Calle Fontana 7384	nulo	19	3.25

Figura 6.1 Atributos y tuplas de una relación ESTUDIANTE.

una relación no tienen un orden específico. En cambio, los registros de un archivo se almacenan físicamente en el disco, de modo que siempre existe un orden entre ellos. Este ordenamiento indica el primero, segundo, i-ésimo y último registros del archivo. De manera similar, cuando presentamos una relación en forma de tabla, las filas se muestran en cierto orden.

El ordenamiento de las tuplas no forma parte de la definición de una relación, porque la relación intenta representar los hechos en un nivel lógico o abstracto. Podemos especificar muchos ordenamientos lógicos en una relación; por ejemplo, las tuplas en la relación ESTUDIANTE de la figura 6.1 se podrían ordenar lógicamente según los valores de Nombre, NSS, Edad o algún otro atributo. La definición de una relación no especifica ningún orden: *no existe preferencia* por ningún ordenamiento lógico en particular. Por tanto, la relación de la figura 6.2 se considera *idéntica* a la de la figura 6.1. Cuando una relación se implementa en forma de archivo, se puede especificar un ordenamiento físico para los registros del archivo.

Orden de los valores dentro de una tupla, y definición alternativa de relación. De acuerdo con la definición anterior de relación, una n-tupla es una *lista ordenada* de n valores, así que el orden de los valores de una tupla —y por tanto de los atributos en la definición de un esquema de relación— es importante. No obstante, en un nivel lógico, el orden de los atributos y de sus valores en realidad *no es* importante en tanto se mantenga la correspondencia entre atributos y valores.

Hay una **definición alternativa de relación** que hace *innecesario* el ordenamiento de los valores en una tupla. Según esta definición, un esquema de relación $R = \{A_1, A_2, ..., A_n\}$ es un *conjunto* de atributos, y una relación $r(R)$ es un conjunto finito de **transformaciones** $r = \{t_1, t_2, ..., t_m\}$, donde cada tupla t_i es una transformación de R a D, y D es la unión de los dominios de los atributos; esto es, $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \cdots \cup \text{dom}(A_n)$. Según esta definición, $t(A_i)$ debe estar en $\text{dom}(A_i)$, con $1 \leq i \leq n$, para cada transformación t en r. Cada transformación t_i se denomina tupla.

De acuerdo con esta definición, podemos considerar una **tupla** como un **conjunto** de pares (<atributo>, <valor>), donde cada par da el valor de la transformación de un atributo A_i y un valor v_i de $\text{dom}(A_i)$. El ordenamiento de los atributos *no es* importante, porque el nombre del atributo aparece junto con su valor. Según esta definición, las dos tuplas de la figura 6.3 son idénticas. Esto tiene sentido en un nivel abstracto o lógico, ya que en realidad no existe ninguna razón para preferir que un valor de atributo aparezca antes que otro en una tupla.

Cuando una relación se implementa en forma de archivo, los atributos pueden ordenarse físicamente como campos dentro de un registro. Usaremos la **primera definición** de relación, donde los atributos y los valores dentro de las tuplas *sí están ordenados*, porque así se simplifica bastante la notación. Sin embargo, la definición alternativa que se dio aquí es más general.

ESTUDIANTE	Nombre	NSS	TelParticular	Dirección	TelOficina	Edad	Prom
	Diego Domínguez	422-11-2320	nulo	Ave. Espina 3452	749-1253	25	3.53
	Bárbara Benet	533-69-1238	839-8461	Calle Fontana 7384	nulo	19	3.25
	Carlos Cortés	489-22-1100	376-9821	Calle Libertad 265	749-6492	28	3.93
	Karla Armenta	381-62-1245	375-4409	Ave. Maíz 125	nulo	18	2.89
	Benjamín Baeza	305-61-2435	373-1616	Calle Balboa 2918	nulo	19	3.21

Figura 6.2 La misma relación ESTUDIANTE de la figura 6.1 con las filas en otro orden.

t = <(Nombre, Diego Domínguez),(NSS, 422-11-2320),(TelParticular, nulo),(Dirección, Ave. Espina3452), (TelOficina, 749-1253),(Edad, 25),(Prom, 3.53)>

t = <(Dirección, Ave. Espina 3452),(Nombre, Diego Domínguez),(NSS, 422-11-2320), (Edad, 25),(TelOficina, 749-1253),(Prom, 3.53),(TelParticular, nulo)>

Figura 6.3 Dos tuplas idénticas cuando el orden de los atributos y de los valores no forma parte de la definición de una relación.

Valores en las tuplas. Cada valor en una tupla es un valor **atómico**; esto es, no es divisible en componentes en lo que respecta al modelo relacional. Por ello no se permiten atributos compuestos ni multivaluados (véase el Cap. 3). Gran parte de la teoría que apoya al modelo relacional se desarrolló tomando en cuenta esta suposición, conocida como suposición de **primera forma normal**. Los atributos multivaluados se deben representar con relaciones individuales, y los atributos compuestos se representan únicamente mediante sus atributos componentes simples. En las investigaciones recientes sobre el modelo relacional se ha intentado eliminar estas restricciones empleando el concepto de relaciones **no en primera forma normal** o **anidadas** (véase Cap. 21).

Puede ser que los valores de algunos atributos dentro de una tupla en particular sean desconocidos o no se apliquen a esa tupla. En estos casos se utiliza un valor especial, llamado **nulo**. Por ejemplo, en la figura 6.1 algunas tuplas de estudiante tienen nulo como teléfono de oficina porque no tienen oficina. Otro estudiante tiene nulo como teléfono particular, probablemente porque no tiene teléfono en su domicilio o porque lo tiene pero no lo sabemos. En general, podemos tener *varios tipos* de valores nulos, como "valor desconocido", "atributo no aplicable a esta tupla" o "esta tupla no tiene valor para este atributo". De hecho, algunas implementaciones establecen diferentes códigos para los distintos tipos de valores nulos. Se ha constatado que es difícil incorporar diferentes tipos de valores nulos en las operaciones del modelo relacional, y un análisis a fondo de este problema rebasa el alcance del presente libro.

Interpretación de una relación. El esquema de una relación se puede interpretar como una declaración o como un tipo de **aserción**. Por ejemplo, el esquema de la relación ESTUDIANTE de la figura 6.1 establece que, en general, una entidad estudiante tiene Nombre, NSS, TelParticular, Dirección, TelOficina, Edad y Prom. Así, cada tupla de la relación se puede interpretar como un **hecho** o un ejemplar particular de la aserción. Por ejemplo, la primera tupla de la figura 6.1 establece el hecho de que existe un ESTUDIANTE cuyo nombre es Benjamín Baeza, cuyo NSS es 305-61-2435, cuya edad es 19, y así sucesivamente.

Cabe señalar que algunas relaciones pueden representar hechos acerca de *entidades*, en tanto que otras pueden representar hechos sobre vínculos. Por ejemplo, un esquema de relación SE_GRADÚA (NSSEstudiante, CódigoDepto) establece que los estudiantes se gradúan en departamentos académicos; una tupla de esta relación relaciona un estudiante con el departamento en que se gradúa. Así pues, el modelo relacional representa hechos acerca de entidades y de vínculos *uniformemente* como relaciones.

Alternativamente, un esquema de relación puede interpretarse como **predicado**; en este caso, los valores de cada tupla se interpretan como valores que *satisfacen* el predicado. Esta interpretación es muy útil en el contexto de los lenguajes de programación lógica, como PROLOG, porque permite usar el modelo relacional dentro de estos lenguajes. Hablaremos más al respecto en el capítulo 24, que trata las bases de datos deductivas.

6.1.3 *Notación del modelo relacional*

Usaremos la siguiente notación en nuestra exposición:

- Un esquema de relación R de grado n se denotará con $R(A_1, A_2, ..., A_n)$.
- Una n-tupla t de una relación $r(R)$ se denotará con $t = <v_1, v_2, ..., v_n>$, donde v_i es el valor que corresponde al atributo A_i. La siguiente notación se refiere a los **valores componentes** de las tuplas:
  - $t[A_i]$ se refiere al valor v_i de t para el atributo A_i.
  - $t[A_u, A_w, ..., A_z]$, donde $A_u, A_w, ..., A_z$ es una lista de atributos de R, se refiere a la subtupla de valores $<v_u, v_w, ..., v_z>$ de t que corresponden a los atributos especificados en la lista.
- Las letras Q, R, S denotan nombres de relaciones.
- Las letras q, r, s denotan estados de relaciones.
- Las letras t, u, v denotan tuplas.
- En general, el nombre de una relación como ESTUDIANTE indica el conjunto actual de tuplas en esa relación —el *estado actual de la relación*, o *ejemplar*— en tanto que ESTUDIANTE(Nombre, NSS, ...) se refiere al esquema de la relación.
- Los nombres de atributos se califican a veces con el nombre de la relación a la que pertenecen; por ejemplo, ESTUDIANTE.Nombre o ESTUDIANTE.Edad.

Consideremos la tupla t = <'Bárbara Benet', '533-69-1238', '839-8461', 'Calle Fontana 7384', nulo, 19, 3.25> de la relación ESTUDIANTE de la figura 6.1; tenemos t[Nombre] = <'Bárbara Benet'> y t[NSS, Prom, Edad] = <'533-69-1238', 3.25, 19>.

6.2 Restricciones del modelo relacional

En esta sección analizaremos los diversos tipos de restricciones que se pueden especificar en un esquema de base de datos relacional. Entre estas restricciones se cuentan las de dominio, de clave, de integridad de entidades y de integridad referencial. Otros tipos de restricciones, llamadas *dependencias de los datos* (que incluyen las *dependencias funcionales* y las *dependencias multivaluadas*) se utilizan principalmente para el diseño de bases de datos por normalización y se verán en los capítulos 12 y 13.

6.2.1 *Restricciones de dominio*

Las restricciones de dominio especifican que el valor de cada atributo A debe ser un valor atómico del dominio dom(A) para ese atributo. Ya vimos, en la sección 6.1.1, las formas de especificar los dominios. Los tipos de datos asociados a los dominios por lo regular incluyen los tipos de datos numéricos estándar de los números enteros (como entero-corto, entero, entero-largo) y reales (flotante y flotante de doble precisión). También disponemos de caracteres, cadenas de longitud fija y cadenas de longitud variable, así como tipos de datos de fecha, hora, marca de tiempo y dinero. Otros dominios posibles se pueden describir mediante

un subintervalo de valores de un tipo de datos o como un tipo de datos enumerado en el que se listan explícitamente todos los valores posibles. En vez de describirlos todos aquí con detalle, en la sección 7.1.2 examinaremos los tipos de datos que ofrece la norma relacional SQL2.

6.2.2 *Restricciones de clave*

Una relación se define como un *conjunto de tuplas*. Por definición, todos los elementos de un conjunto son distintos; por tanto, todas las tuplas de una relación deben ser distintas. Esto significa que no puede haber dos tuplas que tengan la misma combinación de valores para *todos* sus atributos. Por lo regular existen otros **subconjuntos de atributos** de un esquema de relación R con la propiedad de que no debe haber dos tuplas en un ejemplar de relación r de R con la misma combinación de valores para estos atributos. Suponga que denotamos un subconjunto así de atributos con SC; entonces, para cualesquiera dos tuplas distintas t_1 y t_2 en un ejemplar de relación r de R, tenemos la siguiente restricción:

$$t_1[SC] \neq t_2[SC]$$

Todo conjunto de atributos SC de este tipo es una **superclave** del esquema de relación R. Toda relación tiene por lo menos una superclave: el conjunto de todos sus atributos. Sin embargo, una superclave puede tener atributos redundantes, así que un concepto más útil es el de *clave*, que carece de redundancia. Una **clave** K de un esquema de relación R es una superclave de R con la propiedad adicional de que la eliminación de cualquier atributo A de K deja un conjunto de atributos K' que no es una superclave de R. Por tanto, una clave es una *superclave mínima*; una superclave a la cual no podemos quitarle atributos sin que deje de cumplirse la restricción de unicidad.

Como ejemplo consideremos la relación ESTUDIANTE de la figura 6.1. El conjunto de atributos {NSS} es una clave de ESTUDIANTE porque no puede haber dos tuplas de estudiantes que tengan el mismo valor de NSS. Cualquier conjunto de atributos que contenga a NSS —por ejemplo, {NSS, Nombre, Edad}— es una superclave. Sin embargo, la superclave {NSS, Nombre, Edad} no es una clave de ESTUDIANTE, porque si eliminamos Nombre o Edad, o ambos, del conjunto todavía tendremos una superclave.

El valor de un atributo clave puede servir para identificar de manera única una tupla de la relación. Por ejemplo, el valor de NSS 305-61- 2435 identifica de manera única la tupla correspondiente a Benjamín Baeza en la relación ESTUDIANTE. Observe que el hecho de que un conjunto de atributos constituya una clave es una propiedad del esquema de la relación; es una restricción que debe cumplirse en *todos* los ejemplares de relaciones del esquema. La clave se determina a partir del significado de los atributos en el esquema de la relación; por ende, la propiedad *no varía con el tiempo*; debe seguir siendo válida aunque insertemos tuplas nuevas en la relación. Por ejemplo, no podemos ni debemos designar como clave el atributo Nombre de la relación ESTUDIANTE de la figura 6.1, porque no hay garantía de que nunca existirán dos estudiantes con nombres idénticos.†

En general, un esquema de relación puede tener más de una clave. En tal caso, cada una de ellas se denomina **clave candidata**. Por ejemplo, la relación COCHE de la figura 6.4 tiene dos claves candidatas: NúmMatrícula y NúmSerieMotor. Es común designar a una de las claves candidatas como **clave primaria** de la relación. Ésta es la clave candidata cuyos valores

†Hay ocasiones en que los nombres se usan como claves, pero en tal caso se requiere alguna estratagema —como la anexión de un número ordinal— para distinguir entre nombres idénticos.

COCHE	NúmMatrícula	NúmSerieMotor	Marca	Modelo	Año
	Texas ABC-739	A69352	Ford	Mustang	90
	Florida TVP-347	B43696	Oldsmobile	Cutlass	93
	Nueva York MPO-22	X83554	Oldsmobile	Delta	89
	California 432-TFY	C43742	Mercedes	190-D	87
	California RSK-629	Y82935	Toyota	Camry	92
	Texas RSK-629	U028365	Jaguar	XJS	92

Figura 6.4 La relación COCHE con dos claves candidatas: NúmMatrícula y NúmSerieMotor.

sirven para *identificar* las tuplas en la relación. Adoptaremos la convención de subrayar los atributos que forman la clave primaria de un esquema de relación, como en la figura 6.4. Cabe señalar que, cuando un esquema tiene varias claves candidatas, la elección de una para fungir como clave primaria es arbitraria; sin embargo, casi siempre es mejor escoger una clave primaria con un solo atributo o un número reducido de atributos.

6.2.3 *Esquemas de bases de datos relacionales y restricciones de integridad*

Hasta ahora hemos visto relaciones y esquemas de relaciones individuales. Pero, de hecho, una base de datos relacional suele contener muchas relaciones y en éstas las tuplas están relacionadas de diversas maneras. En esta sección definiremos una base de datos relacional

EMPLEADO

NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND

DEPARTAMENTO

NOMBRED	NÚMEROD	NSSGTE	FECHAINICGTE

LUGARES_DEPTOS

NÚMEROD	LUGARD

PROYECTO

NOMBREP	NÚMEROP	LUGARP	NÚMD

TRABAJA_EN

NSSE	NÚMP	HORAS

DEPENDIENTE

NSSE	NOMBRE_DEPENDIENTE	SEXO	FECHAN	PARENTESCO

Figura 6.5 Esquema de la base de datos relacional COMPAÑÍA; las claves primarias están subrayadas.

EMPLEADO

NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
José	B	Silva	123456789	09-ENE-55	Fresnos 731, Higueras, MX	M	30000	333445555	5
Federico	T	Vizcarra	333445555	08-DIC-45	Valle 638, Higueras, MX	M	40000	888665555	5
Alicia	J	Zapata	999887777	19-JUL-58	Catillo 3321, Sucre, MX	F	25000	987654321	4
Jazmín	S	Valdés	987654321	20-JUN-31	Bravo 291, Belén, MX	F	43000	888665555	4
Ramón	K	Nieto	666884444	15-SEP-52	Espiga 875, Heras, MX	M	38000	333445555	5
Josefa	A	Esparza	453453453	31-JUL-62	Rosas 5631, Higueras, MX	F	25000	333445555	5
Ahmed	V	Jabbar	987987987	29-MAR-59	Dalias 980, Higueras, MX	M	25000	987654321	4
Jaime	E	Botello	888665555	10-NOV-27	Sorgo 450, Higueras, MX	M	55000	nulo	1

LUGARES_DEPTOS

NÚMEROD	LUGARD
1	Higueras
4	Santiago
5	Belén
5	Sacramento
5	Higueras

DEPARTAMENTO

NOMBRED	NÚMEROD	NSSGTE	FECHAINICGTE
Investigación	5	333445555	22-MAY-78
Administración	4	987654321	01-ENE-85
Dirección	1	888665555	19-JUN-71

TRABAJA_EN

NSSE	NÚMP	HORAS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	nulo

PROYECTO

NOMBREP	NÚMEROP	LUGARP	NÚMD
ProductoX	1	Belén	5
ProductoY	2	Sacramento	5
ProductoZ	3	Higueras	5
Automatización	10	Santiago	4
Reorganización	20	Higueras	1
Nuevasprestaciones	30	Santiago	4

DEPENDIENTE

NSSE	NOMBRE_DEPENDIENTE	SEXO	FECHAN	PARENTESCO
333445555	Alicia	F	05-ABR-76	HIJA
333445555	Teodoro	M	25-OCT-73	HIJO
333445555	Jobita	F	03-MAY-48	CÓNYUGE
987654321	Abdiel	M	29-FEB-32	CÓNYUGE
123456789	Miguel	M	01-ENE-78	HIJO
123456789	Alicia	F	31-DIC-78	HIJA
123456789	Elizabeth	F	05-MAY-57	CÓNYUGE

Figura 6.6 Ejemplar (estado) de base de datos relacional del esquema COMPAÑÍA.

y un esquema de base de datos relacional. Un **esquema de base de datos relacional** S es un conjunto de esquemas de relaciones $S = \{R_1, R_2, ..., R_m\}$ y un conjunto de **restricciones de integridad** RI. Un **ejemplar de base de datos relacional** BD de S es un conjunto de ejemplares de relaciones BD $= \{r_1, r_2, ..., r_m\}$ tal que cada r_i es un ejemplar de R_i y tal que las relaciones r_i satisfacen las restricciones de integridad especificadas en RI. La figura 6.5 muestra un esquema de base de datos relacional que llamamos COMPAÑÍA, y la figura 6.6 muestra un ejemplar de base de datos relacional que corresponde al esquema COMPAÑÍA. Usaremos este esquema y esta base de datos en el presente capítulo y en los capítulos 7 al 9 para crear ejemplos de consultas en diferentes lenguajes relacionales. Cuando nos refiramos a una base de datos relacional, incluiremos implícitamente tanto su esquema como su ejemplar actual.

En la figura 6.5, el atributo NÚMEROD de DEPARTAMENTO y de LUGARES_DEPTOS representa el mismo concepto del mundo real: el número otorgado a un departamento. Ese mismo concepto se llama ND en EMPLEADO y NÚMD en PROYECTO. Permitiremos que un atributo que represente el mismo concepto del mundo real tenga nombres que pueden o no ser idénticos en diferentes relaciones. De manera similar, permitiremos que atributos que representen diferentes conceptos tengan el mismo nombre en relaciones distintas. Por ejemplo, podríamos haber usado el nombre de atributo NOMBRE tanto para NOMBREP de PROYECTO como para NOMBRED de DEPARTAMENTO; en este caso, tendríamos dos atributos con el mismo nombre pero que representarían conceptos diferentes del mundo real: nombres de proyectos y nombres de departamentos.

En algunas de las *primeras versiones del modelo relacional* se hizo la suposición de que el mismo concepto del mundo real, al representarse con un atributo, tendría nombres *idénticos* en todas las relaciones. Esto crea problemas cuando se usa el mismo concepto del mundo real con diferentes papeles (significados) en la misma relación. Por ejemplo, el concepto de número de seguro social aparece dos veces en la relación EMPLEADO de la figura 6.5: una vez en el papel de número de seguro social del empleado, y otra en el papel de número de seguro social del supervisor. A fin de evitar problemas, les dimos nombres de atributo distintos, NSS y NSSSUPER, respectivamente.

Las restricciones de integridad se especifican en el esquema de una base de datos y se deben cumplir en todos los ejemplares de ese esquema. Además de las restricciones de dominio y de clave, hay otros dos tipos de restricciones en el modelo relacional: integridad de entidades e integridad referencial.

6.2.4 *Integridad de entidades, integridad referencial y claves externas*

La **restricción de integridad de entidades** establece que ningún valor de clave primaria puede ser nulo. Esto es porque el valor de la clave primaria sirve para identificar las tuplas individuales en una relación; el que la clave primaria tenga valores nulos implica que no podemos identificar algunas tuplas. Por ejemplo, si dos o más tuplas tuvieran nulo en su clave primaria, tal vez no podríamos distinguirlas.

Las restricciones de clave y de integridad de entidades se especifican sobre relaciones individuales. La **restricción de integridad referencial** se especifica entre dos relaciones y sirve para mantener la consistencia entre tuplas de las dos relaciones. En términos informales, la restricción de integridad referencial establece que una tupla en una relación que haga referencia a otra relación deberá referirse a una *tupla existente* en esa relación. Por ejemplo, en la figura 6.6 el atributo ND de EMPLEADO da el número del departamento para el cual trabaja cada empleado; por tanto, su valor en cada tupla de EMPLEADO deberá coincidir con el valor de NÚMEROD en alguna tupla de la relación DEPARTAMENTO.

Para dar una definición más formal de integridad referencial primero debemos definir el concepto de clave externa. Las condiciones que debe satisfacer una clave externa (dadas a continuación) especifican una restricción de integridad referencial entre los dos esquemas de relaciones R_1 y R_2. Un conjunto de atributos CE en el esquema de relación R_1 es una **clave externa** de R_1 si satisface las dos reglas siguientes:

1. Los atributos de CE tienen el mismo dominio que los atributos de la clave primaria CP de otro esquema de relación R_2; se dice que los atributos CE **hacen referencia** o **se refieren** a la relación R_2.
2. Un valor de CE en una tupla t_1 de R_1 ocurre como valor de CP en alguna tupla t_2 de R_2 o bien es nulo. En el primer caso, tenemos $t_1[CE] = t_2[CP]$, y decimos que la tupla t_1 **hace referencia** o **se refiere** a la tupla t_2.

En una base de datos con muchas relaciones, suele haber muchas restricciones de integridad referencial. Para especificar dichas restricciones es preciso, primero, comprender con claridad el significado o papel que cada uno de los conjuntos de atributos desempeña en los diversos esquemas de relaciones de la base de datos. Las restricciones de integridad referencial casi siempre surgen de los *vínculos entre las entidades* representadas por los esquemas de relaciones. Por ejemplo, consideremos la base de datos de la figura 6.6. En la relación EMPLEADO, el atributo ND se refiere al departamento para el cual trabaja un empleado; por tanto, designamos a ND como clave externa de EMPLEADO, con referencia a la relación DEPARTAMENTO. Esto significa que un valor de ND en cualquier tupla t_1 de la relación EMPLEADO deberá coincidir con un valor de la clave primaria de DEPARTAMENTO —el atributo NÚMEROD— en alguna tupla t_2 de la relación DEPARTAMENTO, o el valor de ND *puede ser nulo* si el empleado no pertenece a ningún departamento. En la figura 6.6 la tupla del empleado "José Silva" hace referencia a la tupla del departamento "Investigación", con lo que se indica que "José Silva" trabaja para ese departamento.

Cabe señalar que una clave externa puede *hacer referencia a su propia relación*. Por ejemplo, el atributo NSSSUPER de EMPLEADO se refiere al supervisor de un empleado, el cual es otro empleado representado por una tupla de la relación EMPLEADO. Así pues, NSSSUPER es una clave externa que hace referencia a la relación EMPLEADO misma. En la figura 6.6, la tupla del empleado "José Silva" hace referencia a la tupla del empleado "Federico Vizcarra", lo cual indica que "Federico Vizcarra" es el supervisor de "José Silva".

Podemos *representar diagramáticamente las restricciones de integridad referencial* trazando un arco dirigido de cada clave externa a la relación a la cual hace referencia. Para mayor claridad, la punta de la flecha puede apuntar a la clave primaria de la relación referida. La figura 6.7 muestra el esquema de la figura 6.5 con las restricciones de integridad referencial representadas de esta manera.

Debemos especificar todas las restricciones de integridad en el esquema de la base de datos relacional si es que nos interesa mantener dichas restricciones en todos los ejemplares de la base de datos. En consecuencia, en un sistema relacional, el lenguaje de definición de datos (DDL) debe contar con mecanismos para especificar los diversos tipos de restricciones para que el SGBD pueda imponerlas automáticamente. La mayoría de los sistemas de gestión de bases de datos relacionales pueden implantar restricciones de integridad de claves y de entidades, y muchos de ellos ya incorporan mecanismos para establecer la integridad referencial.

Los tipos de restricciones que hemos visto no incluyen una amplia clase de restricciones generales, a veces llamadas *restricciones de integridad semántica*, que puede ser necesario especificar e imponer en una base de datos relacional. Ejemplos de tales restricciones son "el salario de un empleado no debe exceder el salario de su supervisor" y "el número máximo de horas que un empleado puede trabajar en todos los proyectos por semana es 56". Muy pocos de los SGBD relacionales en el mercado apoyan este tipo de restricciones, pero se están desarrollando mecanismos para poder especificarlas e imponerlas.

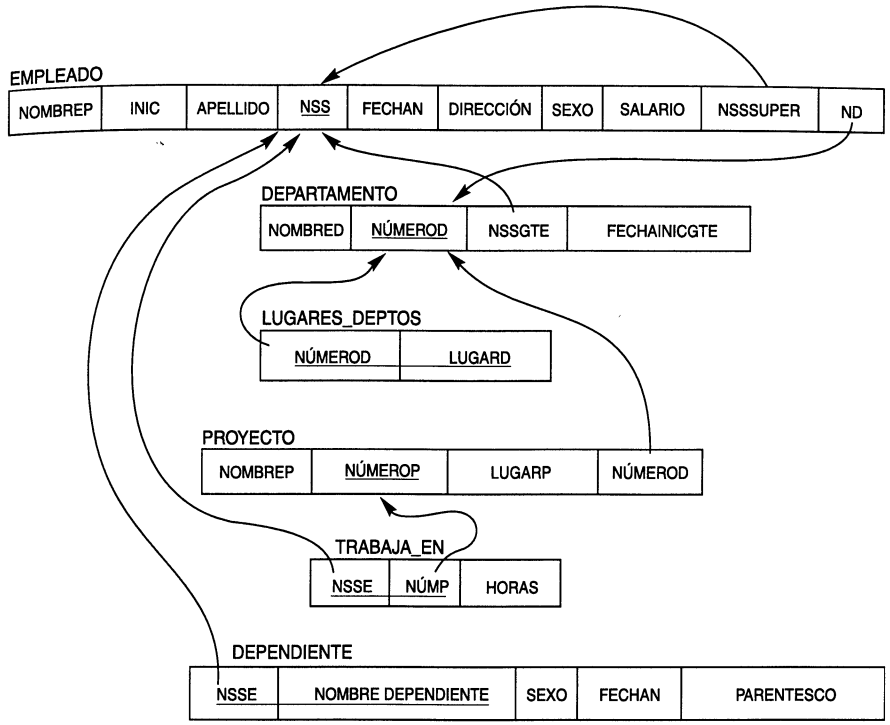


Figura 6.7 Restricciones de integridad referencial representadas en el esquema de la base de datos relacional COMPAÑÍA.

6.3 Operaciones de actualización con relaciones★

Es posible clasificar las operaciones del modelo relacional en obtenciones y actualizaciones. Las operaciones del álgebra relacional, con las que podemos especificar obtenciones, se analizarán con detalle en la sección 6.5. En esta sección, nos concentraremos en las operaciones de actualización. Son tres las operaciones de actualización básicas que se efectúan con relaciones: insertar, eliminar y modificar. **Insertar** sirve para insertar una o más tuplas nuevas en una relación; **eliminar** sirve para eliminar tuplas, y **modificar** sirve para alterar los valores de algunos atributos. Siempre que se apliquen operaciones de actualización, se debe cuidar de no violar las restricciones de integridad especificadas en el esquema de la base de datos relacional. En esta sección estudiaremos los tipos de restricciones que puede violar cada una de las operaciones de actualización y los tipos de acciones que se pueden emprender si una actualización llega a provocar una violación. Usaremos la base de datos de la figura 6.6 para los ejemplos y trataremos únicamente las restricciones de clave, las de integridad de entidades y las de integridad referencial ilustradas en la figura 6.7. Para cada tipo de actualización presentaremos algunos ejemplos de operaciones y analizaremos las restricciones que podría violar cada una de ellas.

6.3.1 *La operación insertar*

La operación **insertar** proporciona una lista de valores de atributos para una nueva tupla *t* que se ha de insertar en una relación *R*. La inserción puede violar cualquiera de los cuatro tipos de restricciones que vimos en la sección anterior. Las restricciones de dominio pueden violarse si se proporciona un valor de atributo que no aparezca en el dominio correspondiente. Las restricciones de clave pueden violarse si un valor clave de la nueva tupla *t* ya existe en otra tupla de la relación *r*(*R*). La integridad de entidades puede violarse si la clave primaria de la nueva tupla *t* es nula. Y la integridad referencial puede violarse si el valor de cualquier clave externa de *t* hace referencia a una tupla que no existe en la relación referida. He aquí algunos ejemplos que ilustran lo anterior:

1. Insertar <'Cecilia', 'F', 'Laguardia', '677678989', '05-ABR-50', 'Calle Viento 6357, Malinalco, MX', F, 28000, nulo, 4> en EMPLEADO.

   — Esta inserción satisface todas las restricciones, así que es aceptable.

2. Insertar <'Alicia', 'J', 'Zapata', '999887777', '05-ABR-50', 'Calle Viento 6357, Malinalco, MX', F, 28000, '987654321', 4> en EMPLEADO.

   — Esta inserción viola la restricción de clave porque ya existe otra tupla con el mismo valor de NSS en la relación EMPLEADO.

3. Insertar <'Cecilia', 'F', 'Laguardia', nulo, '05-ABR-50', 'Calle Viento 6357, Malinalco, MX', F, 28000, nulo, 4> en EMPLEADO.

   — Esta inserción viola la restricción de integridad de entidades (nulo en la clave primaria NSS), de modo que no es aceptable.

4. Insertar <'Cecilia', 'F', 'Laguardia', '677678989', '05-ABR-50', 'Calle Viento 6357, Malinalco, MX', F, 28000, '987654321', 7> en EMPLEADO.

   — Esta inserción viola la restricción de integridad referencial especificada sobre ND porque no existe ninguna tupla de DEPARTAMENTO en la que NÚMEROD = 7.

Si una inserción viola una o más restricciones, disponemos de dos opciones. La primera es *rechazar la inserción*, en cuyo caso sería útil que el SGBD explicara al usuario por qué fue rechazada. La segunda es intentar *corregir la razón por la que se rechazó la inserción*. Por ejemplo, en la operación 3, el SGBD podría pedir al usuario que proporcione un valor para NSS y aceptar la inserción si se introduce un valor de NSS válido. En la operación 4, el SGBD podría pedir al usuario que cambie el valor de ND a algún valor válido (o que lo ponga en nulo), o bien pedirle que inserte una tupla DEPARTAMENTO en la que NÚMEROD = 7, y aceptar la inserción sólo después de haberse aceptado tal operación. Adviértase que, en el segundo caso, la inserción puede **propagarse** en reversa (o **retroceder en cascada**) a la relación EMPLEADO si el usuario intenta insertar una tupla para el departamento 7 con un valor de NSSGTE que no exista en la relación EMPLEADO.

6.3.2 *La operación eliminar*

La operación **eliminar** sólo puede violar la integridad referencial, si las claves externas de otras tuplas de la base de datos hacen referencia a la tupla que se ha de eliminar. Para especificar la eliminación, una condición expresada en términos de los atributos de la relación selecciona la tupla (o tuplas) por eliminar. He aquí algunos ejemplos:

1. Eliminar la tupla TRABAJA_EN con NSSE = '999887777' y NÚMP = 10.

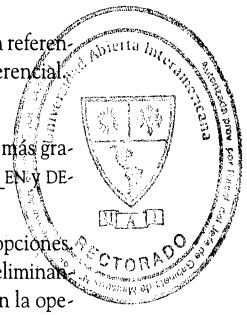
   — Esta eliminación es aceptable.

2. Eliminar la tupla EMPLEADO con NSS = '999887777'.

   — Esta eliminación no es aceptable porque dos tuplas de TRABAJA_EN hacen referencia a esta tupla. Por tanto, si se elimina la tupla, se violará la integridad referencial.

3. Eliminar la tupla EMPLEADO con NSS = '333445555'.

   — Esta eliminación producirá violaciones a la integridad referencial aún más graves, porque tuplas de las relaciones EMPLEADO, DEPARTAMENTO, TRABAJA_EN y DEPENDIENTE hacen referencia a la tupla en cuestión.



Si una operación de eliminación provoca una violación disponemos de tres opciones. La primera es *rechazar la eliminación*. La segunda es *tratar de propagar la eliminación* eliminando las tuplas que hacen referencia a la tupla que se desea eliminar. Por ejemplo, en la operación 2 el SGBD podría eliminar automáticamente las dos tuplas problemáticas de TRABAJA_EN que tienen NSSE = '999887777'. Una tercera opción es *modificar los valores del atributo de referencia* que provocan la violación; todos esos valores se pondrían en nulo o se modificarían de modo que hicieran referencia a otra tupla válida. Nótese que, si un atributo de referencia que origina una violación *forma parte de la clave primaria*, no se puede cambiar a nulo, pues si se hiciera se violaría la integridad de entidades.

Es posible combinar las dos últimas opciones. Por ejemplo, para evitar que la operación 3 cause una violación, el SGBD puede eliminar automáticamente todas las tuplas de TRABAJA_EN y de DEPENDIENTE en las que NSSE = '333445555'. Las tuplas de EMPLEADO en las que NSSSUPER = '333445555' y la tupla de DEPARTAMENTO en la que NSSGTE = '333445555' se pueden eliminar o modificar de modo que tengan valores válidos en NSSSUPER y NSSGTE, o bien el valor nulo. Aunque tal vez sea lógico eliminar automáticamente las tuplas de TRABAJA_EN y de DEPENDIENTE que hacen referencia a una tupla de EMPLEADO, probablemente no sea correcto eliminar otras tuplas de EMPLEADO o una tupla de DEPARTAMENTO. En general, cuando se especifica una restricción de integridad referencial, el SGBD debe permitir al usuario *especificar cuál de las tres opciones* aplicará en caso de violarse la restricción.

6.3.3 *La operación modificar*

La operación **modificar** sirve para cambiar los valores de uno o más atributos en una tupla (o tuplas) de una relación *R*. Es necesario especificar una condición para los atributos de *R* a fin de seleccionar la tupla (o tuplas) que se modificarán. He aquí algunos ejemplos:

1. Modificar el SALARIO de la tupla EMPLEADO con NSS = '999887777' cambiándolo a 28000.

   — Aceptable.

2. Modificar el ND de la tupla EMPLEADO con NSS = '999887777' cambiándolo a 1.

   — Aceptable.

3. Modificar el ND de la tupla EMPLEADO con NSS = '999887777' cambiándolo a 7.

   — Inaceptable, porque viola la integridad referencial.

4. Modificar el NSS de la tupla EMPLEADO con NSS = '999887777' cambiándolo a '987654321'.

   — Inaceptable, porque viola las restricciones de clave primaria y de integridad referencial.

La modificación de un atributo que no es clave primaria ni clave externa; casi nunca causa problemas; basta con que el SGBD constate que el nuevo valor sea del tipo de datos correcto y esté en el dominio. Modificar un valor de clave primaria es similar a eliminar una tupla e insertar otra en su lugar, porque usamos la clave primaria para identificar las tuplas. Por tanto, los problemas que ya vimos al hablar de inserciones y eliminaciones se pueden presentar en este caso también. Si se modifica un atributo de clave externa, el SGBD debe asegurarse de que el nuevo valor haga referencia a una tupla existente en la relación referida.

6.4 Definición de relaciones*

Cuando se desea implementar una base de datos relacional para una aplicación compleja, los diseñadores suelen empezar por *diseñar* con mucho cuidado el esquema de la base de datos. Esto implica decidir cuáles atributos deben ir juntos en cada relación, elegir nombres apropiados para las relaciones y sus atributos, especificar los dominios y tipos de datos de los diversos atributos, identificar las claves candidatas y escoger una clave primaria para cada relación, y especificar todas las claves externas. Analizaremos dos técnicas para diseñar bases de datos relacionales. En la sección 6.8 mostraremos cómo se puede diseñar un esquema de base de datos relacional mediante una transformación de un diseño conceptual cuya creación se basa en el modelo ER (véase el Cap. 3). En los capítulos 12 y 13 describiremos la teoría de la normalización y los algoritmos de diseño relacional basados en las dependencias funcionales y multivaluadas. En esta sección, supondremos que ya se ha diseñado un esquema de base de datos relacional y veremos cómo los usuarios pueden declarar las relaciones individuales.

Todo SGBD relacional debe contar con un lenguaje de definición de datos (DDL) para definir los esquemas de relaciones. La mayoría de los DDL se basan en el lenguaje SQL, y en la sección 7.1 presentaremos el DDL de SQL. Aquí analizaremos los componentes (idealizados) del lenguaje que se necesitan para declarar un esquema de relación. El primer paso es dar un nombre al esquema completo de la base de datos relacional para poder asignarle relaciones individuales, mediante una declaración como ésta:

```
DECLARE SCHEMA COMPAÑÍA;
```

El siguiente paso consiste en declarar los dominios que requieren los atributos, dando a cada dominio un nombre y un tipo de datos. Declararemos los posibles dominios para los atributos de las relaciones EMPLEADO y DEPARTAMENTO de la figura 6.7 así:

```
DECLARE DOMAIN NSS_PERSONAS TYPE FIXED_CHAR (9);
DECLARE DOMAIN NOMBRES_PERSONAS TYPE VARIABLE_CHAR (15);
DECLARE DOMAIN INICIALES_PERSONAS TYPE ALPHABETIC_CHAR (1);
DECLARE DOMAIN FECHAS TYPE DATE;
DECLARE DOMAIN DIRECCIONES TYPE VARIABLE_CHAR (35);
DECLARE DOMAIN SEXO_PERSONAS TYPE ENUMERATED {M, F};
DECLARE DOMAIN SALARIOS_PERSONAS TYPE MONEY;
```



```
DECLARE DOMAIN NÚMEROS_DEPTOS TYPE INTEGER_RANGE [1, 10];
DECLARE DOMAIN NOMBRES_DEPTOS TYPE VARIABLE_CHAR (20);
```

Ahora podemos definir las relaciones individuales. Se requieren elementos para especificar el nombre de la relación, los nombres de los atributos y dominios, las claves primarias y de otro tipo, y las claves externas. Para declarar las relaciones EMPLEADO y DEPARTAMENTO de la figura 6.7 podemos usar declaraciones como éstas:

```
DECLARE RELATION EMPLEADO
FOR SCHEMA  COMPAÑÍA
ATTRIBUTES  NOMBREP      DOMAIN NOMBRES_PERSONAS,
            INICIAL      DOMAIN INICIALES_PERSONAS,
            APELLIDO     DOMAIN NOMBRES_PERSONAS,
            NSS          DOMAIN NSS_PERSONAS,
            FECHAN       DOMAIN FECHAS,
            DIRECCIÓN    DOMAIN DIRECCIONES,
            SEXO         DOMAIN SEXO_PERSONAS,
            SALARIO      DOMAIN SALARIOS_PERSONAS,
            NSSSUPER     DOMAIN NSS_PERSONAS,
            ND           DOMAIN NÚMEROS_DEPTOS
CONSTRAINTS PRIMARY_KEY (NSS),
            FOREIGN_KEY (NSSSUPER) REFERENCES EMPLEADO,
            FOREIGN_KEY (ND) REFERENCES DEPARTAMENTO;

DECLARE RELATION DEPARTAMENTO
FOR SCHEMA  COMPAÑÍA
ATTRIBUTES  NOMBRED      DOMAIN NOMBRES_DEPTOS,
            NÚMEROD      DOMAIN NÚMEROS_DEPTOS,
            NSSGTE       DOMAIN NSS_PERSONAS,
            FECHAINICGTE DOMAIN FECHAS
CONSTRAINTS PRIMARY_KEY (NÚMEROD),
            KEY (NOMBRED),
            FOREIGN_KEY (NSSGTE) REFERENCES EMPLEADO;
```

Con estos ejemplos hemos ofrecido una breve introducción a los elementos que se requieren en un DDL relacional. En el capítulo 7 veremos los elementos principales del DDL de SQL, en el que se basan casi todos los SGBD relacionales que encontramos en el mercado.

6.5 El álgebra relacional

Hasta aquí sólo hemos examinado los conceptos para definir la estructura y las restricciones de una base de datos, en el modelo relacional, y para ejecutar las operaciones relacionales de actualización. Ahora dirigiremos nuestra atención al álgebra relacional: una colección de operaciones que sirven para manipular relaciones enteras. Estas operaciones sirven, por ejemplo, para seleccionar tuplas de relaciones individuales y para combinar tuplas relacionadas a partir de varias relaciones con el fin de especificar una consulta —una solicitud de obtención— de la base de datos. El resultado de cada operación es una nueva relación, que podremos manipular en una ocasión futura.

Las operaciones del álgebra relacional suelen clasificarse en dos grupos. Uno contiene las operaciones corrientes de la teoría matemática de conjuntos; es posible aplicarlas porque las relaciones se definen como conjuntos de tuplas. Entre las operaciones de conjuntos están la UNIÓN, la INTERSECCIÓN, la DIFERENCIA y el PRODUCTO CARTESIANO. El otro grupo consiste en operaciones creadas específicamente para bases de datos relacionales; incluyen SELECCIONAR, PROYECTAR y REUNIÓN (a esta última también le llaman JUNTA), entre otras. Primero veremos las operaciones SELECCIONAR y PROYECTAR porque son las más sencillas; luego estudiaremos las operaciones de conjuntos. Por último, trataremos la REUNIÓN y otras operaciones complejas. Para nuestros ejemplos nos apoyaremos en la base de datos relacional de la figura 6.6.

6.5.1 *La operación* SELECCIONAR

La operación **SELECCIONAR** sirve para seleccionar un *subconjunto* de las tuplas de una relación que satisfacen una **condición de selección**. Por ejemplo, para seleccionar el subconjunto de tuplas de EMPLEADO que trabajan en el departamento 4 o cuyo salario rebasa los $30 000, podemos especificar individualmente cada una de estas dos condiciones con la operación SELECCIONAR, como sigue:

$\sigma_{ND=4}(\text{EMPLEADO})$
$\sigma_{SALARIO>30000}(\text{EMPLEADO})$

En general, denotamos la operación SELECCIONAR con

$\sigma_{<\text{condición de selección}>}(<\text{nombre de la relación}>)$

donde el símbolo σ (sigma) denota el operador de SELECCIONAR, y la condición de selección es una expresión booleana especificada en términos de los atributos de la relación.

La relación que resulta de la operación SELECCIONAR tiene los *mismos atributos* que la relación especificada en <nombre de la relación>. La expresión booleana especificada en la <condición de selección> se compone de una o más **cláusulas** de la forma:

<nombre de atributo> <operador de comparación> <valor constante>, o
<nombre de atributo> <operador de comparación> <nombre de atributo>

donde <nombre de atributo> es el nombre de un atributo de <nombre de la relación>, <operador de comparación> es normalmente uno de los operadores $\{=, <, \leq, >, \geq, \neq\}$, y <valor constante> es un valor constante del dominio del atributo. Las cláusulas pueden conectarse arbitrariamente con los operadores booleanos Y (AND), O (OR) y NO (NOT) para formar una condición de selección general. Por ejemplo, si queremos seleccionar las tuplas de todos los empleados que trabajan en el departamento 4 y ganan más de $25 000 al año, o que trabajan en el departamento 5 y ganan más de $30 000, podemos especificar la siguiente operación SELECCIONAR:

$\sigma_{(ND=4\ Y\ SALARIO>25000)\ O\ (ND=5\ Y\ SALARIO>30000)}(\text{EMPLEADO})$

El resultado se muestra en la figura 6.8(a).

Cabe señalar que los operadores de comparación del conjunto $\{=, <, \leq, >, \geq, \neq\}$ se aplican a atributos cuyos dominios son *valores ordenados*, como los dominios numéricos o de fechas. Los dominios de cadenas de caracteres se consideran ordenados con base en la secuencia de cotejo de los caracteres. Si el dominio de un atributo es un conjunto de *valores no ordenados*, sólo se podrán aplicar los operadores de comparación del conjunto $\{=, \neq\}$ a

(a)

NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
Federico	T	Vizcarra	333445555	08-DIC-45	Valle 638, Higueras, MX	M	40000	888665555	5
Jazmín	S	Valdés	987654321	20-JUN-31	Bravo 291, Belén, MX	F	43000	888665555	4
Ramón	K	Nieto	666884444	15-SEP-52	Espiga 975, Heras, MX	M	38000	333445555	5

(b)

APELLIDO	NOMBREP	SALARIO
Silva	José	30000
Vizcarra	Federico	40000
Zapata	Alicia	25000
Valdés	Jazmín	43000
Nieto	Ramón	38000
Esparza	Josefa	25000
Jabbar	Ahmed	25000
Botello	Jaime	55000

(c)

SEXO	SALARIO
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

Figura 6.8 Resultados de operaciones SELECCIONAR y PROYECTAR.
(a) $\sigma_{(ND=4\ Y\ SALARIO>25000)\ O\ (ND=5\ Y\ SALARIO>30000)}(\text{EMPLEADO})$
(b) $\pi_{APELLIDO,\ NOMBREP,\ SALARIO}(\text{EMPLEADO})$
(c) $\pi_{SEXO,\ SALARIO}(\text{EMPLEADO})$

ese atributo. Un ejemplo de dominio no ordenado es el dominio Color = {rojo, azul, verde, blanco, amarillo, ...}, donde no se especifica orden alguno entre los diferentes colores. Algunos dominios permiten tipos adicionales de operadores de comparación; por ejemplo, en un dominio de cadenas de caracteres podríamos contar con el operador de comparación SUBCADENA_DE.

En general, el resultado de una operación SELECCIONAR se determina como sigue. Se aplica la <condición de selección> independientemente a cada tupla t en la relación R especificada por <nombre de la relación>. Esto se hace sustituyendo cada ocurrencia de un atributo A_i en la condición de selección por su valor en la tupla $t[A_i]$. Si el resultado de evaluar la condición es "verdadero", se **seleccionará** la tupla t. Todas las tuplas seleccionadas aparecen en el resultado de la operación SELECCIONAR. Las condiciones booleanas Y (AND), O (OR) y NO (NOT) se interpretan de la manera normal, a saber:

- (cond1 Y cond2) es verdadera si tanto (cond1) como (cond2) son verdaderas; en caso contrario, es falsa.
- (cond1 O cond2) es verdadera si (cond1) o (cond2), o ambas, son verdaderas; en caso contrario, es falsa.
- (NO cond) es verdadera si cond es falsa; en caso contrario, es falsa.

El operador SELECCIONAR es unario; esto es, se aplica a una sola relación. Por ello, no podemos usar SELECCIONAR para seleccionar tuplas de más de una relación. Por añadidura, la operación de selección se aplica a *cada tupla individualmente*; por tanto, las condiciones de selección no pueden abarcar más de una tupla. El **grado** de la relación resultante de una operación SELECCIONAR es el mismo que el de la relación original R a la que se aplicó la operación, porque tiene los mismos atributos que R. El número de tuplas de la relación resultante siempre es *menor que* el número de tuplas de la relación original R *o igual a* él. La fracción de las tuplas seleccionadas por una condición de selección se denomina **selectividad** de la condición.

Observe que la operación SELECCIONAR es **conmutativa**; es decir,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(R)) = \sigma_{<cond2>}(\sigma_{<cond1>}(R))$$

Así pues, podemos aplicar una secuencia de operaciones SELECCIONAR en cualquier orden. Además, siempre podemos combinar una **cascada** de operaciones SELECCIONAR en una sola operación SELECCIONAR con una condición conjuntiva (Y); es decir,

$$\sigma_{<cond1>}(\sigma_{<cond2>}(\ldots(\sigma_{<condn>}(R))\cdots)) = \sigma_{<cond1> \text{ Y } <cond2> \text{ Y } \ldots \text{ Y } <condn>}(R)$$

6.5.2 *La operación* PROYECTAR

Si visualizamos una relación como una tabla, la operación SELECCIONAR selecciona algunas *filas* de la tabla y desecha otras. La operación **PROYECTAR**, en cambio, selecciona ciertas *columnas* de la tabla y desecha las demás. Si sólo nos interesan ciertos atributos de una relación, "proyectaremos" la relación sobre esos atributos con la operación PROYECTAR. Por ejemplo, si queremos listar el apellido, el nombre de pila y el salario de todos los empleados, podemos usar la siguiente operación PROYECTAR:

$$\pi_{APELLIDO,NOMBREP,\ SALARIO}(EMPLEADO)$$

La relación resultante se muestra en la figura 6.8(b). La forma general de la operación PROYECTAR es

$$\pi_{<lista\ de\ atributos>}(<nombre\ de\ la\ relación>)$$

donde π (pi) es el símbolo para la operación PROYECTAR y <lista de atributos> es una lista de atributos de la relación especificada por <nombre de la relación>. La relación así creada tiene sólo los atributos especificados en <lista de atributos> y *en el mismo orden en que aparecen en la lista*. Por ello, su **grado** es igual al número de atributos en <lista de atributos>.

Si la lista de atributos sólo contiene atributos no clave de una relación, es probable que aparezcan tuplas repetidas en el resultado. La operación PROYECTAR *elimina* implícitamente *cualesquier tuplas repetidas*, así que el resultado de la operación PROYECTAR es un conjunto de tuplas y por tanto una relación válida. Por ejemplo, consideremos la siguiente operación PROYECTAR:

$$\pi_{SEXO,\ SALARIO}(EMPLEADO)$$

El resultado se muestra en la figura 6.8(c). La tupla <F, 25000> sólo aparece una vez en dicha figura, aunque su combinación de valores aparece dos veces en la relación EMPLEADO. Siempre que aparezcan dos o más tuplas idénticas al aplicar una operación PROYECTAR, sólo una se conservará en el resultado; esto se conoce como **eliminación de duplicados** y es necesaria para garantizar que el resultado de la operación PROYECTAR sea también una relación: un *conjunto* de tuplas.

El número de tuplas en una relación que resulta de una operación PROYECTAR siempre es menor que el número de tuplas de la relación original, o igual a él. Si la lista de proyección incluye una clave de la relación, la relación resultante tendrá el *mismo número* de tuplas que la original. Por añadidura,

$$\pi_{<lista1>}(\pi_{<lista2>}(R)) = \pi_{<lista1>}(R)$$

siempre que <lista2> contenga los atributos que están en <lista1>; si no es así, el lado izquierdo será incorrecto. Vale la pena señalar también que la operación PROYECTAR *no es* conmutativa.

6.5.3 *Secuencias de operaciones y cambio de nombre de los atributos*

Las relaciones que aparecen en la figura 6.8 carecen de nombres. En general, es posible que deseemos aplicar varias operaciones del álgebra relacional una tras otra. Para ello, podemos escribir las operaciones en una sola **expresión del álgebra relacional** anidándolas, o bien podemos aplicar una operación a la vez y crear relaciones de resultados intermedios. En el segundo caso, tendremos que nombrar las relaciones que contienen los resultados intermedios. Por ejemplo, si queremos obtener el nombre de pila, el apellido y el salario de todos los empleados que trabajan en el departamento número 5, deberemos aplicar una operación SELECCIONAR y una operación PROYECTAR. Podemos escribir una sola expresión del álgebra relacional, a saber:

$$\pi_{NOMBREP,\ APELLIDO,\ SALARIO}(\sigma_{ND=5}(EMPLEADO))$$

La figura 6.9(a) muestra el resultado que arroja esta expresión del álgebra relacional. Como alternativa, podemos mostrar explícitamente la secuencia de operaciones, dando un nombre a cada una de las relaciones intermedias, como sigue:

$$EMPS\_DEP5 \leftarrow \sigma_{ND=5}(EMPLEADO)$$
$$RESULTADO \leftarrow \pi_{NOMBREP,\ APELLIDO,\ SALARIO}(EMPS\_DEP5)$$

A menudo es más sencillo descomponer una secuencia compleja de operaciones especificando relaciones de resultados intermedios que escribiendo una sola expresión del álgebra relacional. También podemos usar esta técnica para **cambiar el nombre de los atributos** de las relaciones intermedias y de la resultante. Esto puede ser útil cuando se trata de operaciones

(a)

NOMBREP	APELLIDO	SALARIO
José	Silva	30000
Federico	Vizcarra	40000
Ramón	Nieto	38000
Josefa	Esparza	25000

(b)

TEMP	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	José	B	Silva	123456789	09-ENE-55	Fresnos 731, Higueras, MX	M	30000	333445555	5
	Federico	T	Vizcarra	333445555	08-DIC-45	Valle 638, Higueras, MX	M	40000	888665555	5
	Ramón	K	Nieto	666884444	15-SEP-52	Espiga 975, Heras, MX	M	38000	333445555	5
	Josefa	A	Esparza	453453453	31-JUL-62	Rosas 5631, Higueras, MX	F	25000	333445555	5

R	NOMPILA	APPATERNO	SALARIO
	José	Silva	30000
	Federico	Vizcarra	40000
	Ramón	Nieto	38000
	Josefa	Esparza	25000

Figura 6.9 Resultados de expresiones del álgebra relacional. (a) $\pi_{NOMBREP,\ APELLIDO,\ SALARIO}(\sigma_{ND=5}(EMPLEADO))$. (b) La misma expresión empleando relaciones intermedias y cambiando el nombre de los atributos.

más complejas como UNIÓN y REUNIÓN, como veremos más adelante. Presentaremos aquí la notación para cambiar los nombres. Si queremos cambiar los nombres de los atributos de una relación que resulte de aplicar una operación del álgebra relacional, bastará con que incluyamos una lista con los nuevos nombres de atributos entre paréntesis, como en el siguiente ejemplo:

$$\text{TEMP} \leftarrow \sigma_{\text{ND=5}}(\text{EMPLEADO})$$
$$\text{R(NOMPILA, APPATERNO, SALARIO)} \leftarrow \pi_{\text{NOMBREP, APELLIDO, SALARIO}}(\text{TEMP})$$

Las dos operaciones anteriores se ilustran en la figura 6.9(b). Si no se cambian los nombres, los atributos de la relación resultante de una operación SELECCIONAR serán los mismos que los de la relación original y estarán en el mismo orden. En el caso de una operación PROYECTAR sin cambio de nombres, la relación resultante tendrá los mismos nombres de atributos que aparecen en la lista de proyección, y en el mismo orden.

6.5.4 *Operaciones de la teoría de conjuntos*

El siguiente grupo de operaciones del álgebra relacional son las operaciones matemáticas normales de conjuntos. Se aplican al modelo relacional porque las relaciones se definen como conjuntos de tuplas, y pueden servir para procesar las tuplas de dos relaciones como conjuntos. Por ejemplo, si queremos obtener los números de seguro social de todos los empleados que trabajan en el departamento 5 o que supervisan directamente a un empleado que trabaja en ese mismo departamento, podemos utilizar la operación UNIÓN como sigue:

$$\text{EMPS\_DEP5} \leftarrow \sigma_{\text{ND=5}}(\text{EMPLEADO})$$
$$\text{RESULTADO1} \leftarrow \pi_{\text{NSS}}(\text{EMPS\_DEP5})$$
$$\text{RESULTADO2(NSS)} \leftarrow \pi_{\text{NSSSUPER}}(\text{EMPS\_DEP5})$$
$$\text{RESULTADO} \leftarrow \text{RESULTADO1} \cup \text{RESULTADO2}$$

La relación RESULTADO1 contiene los números de seguro social de todos los empleados que trabajan en el departamento 5, y RESULTADO2 contiene los números de seguro social de todos los empleados que supervisan directamente a empleados que trabajan en el departamento 5. La operación de UNIÓN produce las tuplas que están en RESULTADO1 o en RESULTADO2, o en ambas (véase la Fig. 6.10).

Se utilizan varias operaciones de la teoría de conjuntos para combinar de diversas maneras los elementos de dos conjuntos, entre ellas, UNIÓN, INTERSECCIÓN y DIFERENCIA. Estas operaciones son binarias; es decir, se aplican a dos conjuntos. Al adaptar estas operaciones a las bases de datos relacionales, debemos asegurarnos de que se puedan aplicar a dos relaciones para que el resultado también sea una relación válida. Para ello, las dos relaciones a las que se aplique cualquiera de las tres operaciones anteriores deberán tener el mismo **tipo de tuplas**; esta condición se denomina *compatibilidad de unión*. Se dice que dos relaciones

RESULTADO1	NSS
	123456789
	333445555
	666884444
	453453453

RESULTADO2	NSS
	333445555
	888665555

RESULTADO	NSS
	123456789
	333445555
	666884444
	453453453
	888665555

Figura 6.10 RESULTADO ← RESULTADO1 ∪ RESULTADO2.

$R(A_1, A_2, ..., A_n)$ y $S(B_1, B_2, ..., B_n)$ son **compatibles con la unión** si tienen el mismo grado n y si $\text{dom}(A_i) = \text{dom}(B_i)$ para $1 \leq i \leq n$. Esto significa que las dos relaciones tienen el mismo número de atributos y que cada par de atributos correspondientes tienen el mismo dominio.

Podemos definir las tres operaciones UNIÓN, INTERSECCIÓN y DIFERENCIA para dos relaciones compatibles con la unión, R y S, como sigue:

- *UNIÓN:* El resultado de esta operación, denotado por $R \cup S$, es una relación que incluye todas las tuplas que están en R o en S o en ambas. Las tuplas repetidas se eliminan.
- *INTERSECCIÓN:* El resultado de esta operación, denotado por $R \cap S$, es una relación que incluye las tuplas que están tanto en R como en S.
- *DIFERENCIA:* El resultado de esta operación, denotado por $R - S$, es una relación que incluye todas las tuplas que están en R pero no en S.

(a)

ESTUDIANTE	NP	AP
	Susana	Yáñez
	Ramón	Sánchez
	Josué	Landa
	Bárbara	Jaimes
	Amanda	Flores
	Jaime	Vélez
	Ernesto	Gómez

PROFESOR	NOMBREP	APELLIDO
	José	Silva
	Ricardo	Bueno
	Susana	Yáñez
	Francisco	Jiménez
	Ramón	Sánchez

(b)

NP	AP
Susana	Yáñez
Ramón	Sánchez
Josué	Landa
Bárbara	Jaimes
Amanda	Flores
Jaime	Vélez
Ernesto	Gómez
José	Silva
Ricardo	Bueno
Francisco	Jiménez

(c)

NP	AP
Susana	Yáñez
Ramón	Sánchez

(d)

NP	AP
Josué	Landa
Bárbara	Jaimes
Amanda	Flores
Jaime	Vélez
Ernesto	Gómez

(e)

NOMBREP	APELLIDO
José	Silva
Ricardo	Bueno
Francisco	Jiménez

Figura 6.11 Las operaciones de conjuntos UNION, INTERSECCION y DIFERENCIA. (a) Dos relaciones compatibles con la union. (b) ESTUDIANTE ∪ PROFESOR. (c) ESTUDIANTE ∩ PROFESOR. (d) ESTUDIANTE — PROFESOR. (e) PROFESOR — ESTUDIANTE.

Adoptaremos la convención de que la relación resultante tiene los mismos nombres de atributos que la *primera* relación R. La figura 6.11 ilustra las tres operaciones. Las relaciones ESTUDIANTE y PROFESOR de la figura 6.11(a) son compatibles con la unión, y sus tuplas representan los nombres de estudiantes y profesores, respectivamente. El resultado de la operación UNIÓN (Fig. 6.11(b)) muestra los nombres de todos los estudiantes y profesores. Recuerde que las tuplas repetidas aparecen sólo una vez en el resultado. El resultado de la operación INTERSECCIÓN (Fig. 6.11(c)) incluye sólo las personas que son al mismo tiempo estudiantes y profesores. Cabe señalar que tanto la UNIÓN como la INTERSECCIÓN son *operaciones conmutativas*; es decir,

$$R \cup S = S \cup R, \text{ y } R \cap S = S \cap R$$

Ambas operaciones pueden aplicarse a *cualquier número de relaciones*, y las dos son *operaciones asociativas*; esto es,

$$R \cup (S \cup T) = (R \cup S) \cup T, \text{ y}$$
$$(R \cap S) \cap T = R \cap (S \cap T)$$

La operación DIFERENCIA *no es conmutativa*; es decir, en general,

$$R - S \neq S - R$$

La figura 6.11(d) muestra los nombres de los estudiantes que no son profesores, y la figura 6.11(e) muestra los nombres de los profesores que no son estudiantes.

A continuación hablaremos del PRODUCTO CARTESIANO, denotado por X. También ésta es una operación binaria de conjuntos, pero las relaciones a las que se aplica *no* tienen que ser compatibles con la unión. Esta operación, conocida también como PRODUCTO CRUZADO o REUNIÓN CRUZADA sirve para combinar tuplas de dos relaciones para poder identificar las tuplas relacionadas entre sí. En general, el resultado de $R(A_1, A_2, ..., A_n)$ X $S(B_1, B_2, ..., B_m)$ es una relación Q con $n + m$ atributos $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, en ese orden. La relación resultante Q tiene una tupla por cada combinación de tuplas: una de R y una de S. Por tanto, si R tiene n_R tuplas y S tiene n_S tuplas, R X S tendrá $n_R * n_S$ tuplas. Para ilustrar el empleo del PRODUCTO CARTESIANO, suponga que deseamos obtener una lista de los dependientes de cada uno de los empleados de sexo femenino. Esto lo podemos lograr como sigue:

EMPS_FEM $\leftarrow \sigma_{SEXO='F'}$(EMPLEADO)
NOMBRESEMP $\leftarrow \pi_{NOMBREP, APELLIDO, NSS}$(EMPS_FEM)
DEPENDIENTES_EMP $\leftarrow$ NOMBRESEMP X DEPENDIENTE
DEPENDIENTES_REALES $\leftarrow \sigma_{NSS=NSSE}$(DEPENDIENTES_EMP)
RESULTADO $\leftarrow \pi_{NOMBREP, APELLIDO, NOMBRE\_DEPENDIENTE}$(DEPENDIENTES_REALES)

Las relaciones que resultan de esta secuencia de operaciones se muestran en la figura 6.12. La relación DEPENDIENTES_EMP es el resultado de aplicar la operación PRODUCTO CARTESIANO a NOMBRESEMP de la figura 6.12 y DEPENDIENTE de la figura 6.6. En DEPENDIENTES_EMP, cada una de las tuplas de NOMBRESEMP se combina con cada una de las tuplas de DEPENDIENTE, lo que da un resultado que no nos dice mucho. Sólo queremos combinar una tupla de empleado de sexo femenino con sus dependientes, a saber, las tuplas DEPENDIENTE cuyo valor de NSSE coincida con el valor de NSS de la tupla EMPLEADO. Esto lo logra la relación DEPENDIENTES_REALES.

El PRODUCTO CARTESIANO crea tuplas con los atributos combinados de dos relaciones. Después podremos seleccionar sólo las tuplas relacionadas de las dos relaciones especificando una condición de selección apropiada, como hicimos en el ejemplo anterior.

EMPS_FEM

NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
Alicia	J	Zapata	999887777	19-JUL-58	Castillo 3321, Sucre, MX	F	25000	987654321	4
Jazmín	S	Valdés	987654321	20-JUN-31	Bravo 291, Belén, MX	F	43000	888665555	5
Josefa	A	Esparza	453453453	31-JUL-62	Rosas 5631, Higueras, MX	F	25000	333445555	5

NOMBRESEMP

NOMBREP	APELLIDO	NSS
Alicia	Zapata	999887777
Jazmín	Valdés	987654321
Josefa	Esparza	453453453

DEPENDIENTES_EMP

NOMBREP	APELLIDO	NSS	NSSE	NOMBRE_DEPENDIENTE	SEXO	FECHAN	...
Alicia	Zapata	999887777	333445555	Alicia	F	05-ABR-76	...
Alicia	Zapata	999887777	333445555	Teodoro	M	25-OCT-73	...
Alicia	Zapata	999887777	333445555	Jobita	F	03-MAY-48	...
Alicia	Zapata	999887777	987654321	Abdiel	M	29-FEB-32	...
Alicia	Zapata	999887777	123456789	Miguel	M	01-ENE-78	...
Alicia	Zapata	999887777	123456789	Alicia	F	31-DIC-78	...
Alicia	Zapata	999887777	123456789	Elizabeth	F	05-MAY-57	...
Jazmín	Valdés	987654321	333445555	Alicia	F	05-ABR-76	...
Jazmín	Valdés	987654321	333445555	Teodoro	M	25-OCT-73	...
Jazmín	Valdés	987654321	333445555	Jobita	F	03-MAY-48	...
Jazmín	Valdés	987654321	987654321	Abdiel	M	29-FEB-32	...
Jazmín	Valdés	987654321	123456789	Miguel	M	01-ENE-78	...
Jazmín	Valdés	987654321	123456789	Alicia	F	31-DIC-78	...
Jazmín	Valdés	987654321	123456789	Elizabeth	F	05-MAY-57	...
Josefa	Esparza	453453453	333445555	Alicia	F	05-ABR-76	...
Josefa	Esparza	453453453	333445555	Teodoro	M	25-OCT-73	...
Josefa	Esparza	453453453	333445555	Jobita	F	03-MAY-48	...
Josefa	Esparza	453453453	987654321	Abdiel	M	29-FEB-32	...
Josefa	Esparza	453453453	123456789	Miguel	M	01-ENE-78	...
Josefa	Esparza	453453453	123456789	Alicia	F	31-DIC-78	...
Josefa	Esparza	453453453	123456789	Elizabeth	F	05-MAY-57	...

DEPENDIENTES_REALES

NOMBREP	APELLIDO	NSS	NSSE	NOMBRE_DEPENDIENTE	SEXO	FECHAN
Jazmín	Valdés	987654321	987654321	Abdiel	M	29-FEB-32

RESULTADO

NOMBREP	APELLIDO	NOMBRE_DEPENDIENTE
Jazmín	Valdés	Abdiel

Figura 6.12 La operación PRODUCTO CARTESIANO.

Como esta secuencia de PRODUCTO CARTESIANO seguido de SELECCIONAR se utiliza con mucha frecuencia para identificar y seleccionar tuplas relacionadas de dos relaciones, se creó una operación especial, llamada REUNIÓN, con el fin de especificar esta secuencia con una sola operación. Hablaremos de la operación REUNIÓN en la siguiente sección. El PRODUCTO CARTESIANO casi nunca se usa como operación significativa por sí sola.

6.5.5 *La operación* REUNIÓN

La operación **REUNIÓN**, denotada por ⋈, sirve para combinar *tuplas relacionadas* de dos relaciones en una sola tupla. Esta operación es muy importante en cualquier base de datos relacional que comprenda más de una relación, porque permite procesar los vínculos entre las relaciones. A fin de ilustrar la reunión, supongamos que deseamos *obtener el nombre del gerente de cada uno de los departamentos*. Para obtenerlo, necesitamos combinar cada tupla de departamento con la tupla de empleado cuyo valor de NSS coincida con el valor de NSSGTE de la tupla de departamento. Esto se logra aplicando la operación REUNIÓN y proyectando el resultado sobre los atributos requeridos:

$$\text{GTE\_DEPTO} \leftarrow \text{DEPARTAMENTO} \bowtie_{\text{NSSGTE=NSS}} \text{EMPLEADO}$$
$$\text{RESULTADO} \leftarrow \pi_{\text{NOMBRED, APELLIDO, NOMBREP}}(\text{GTE\_DEPTO})$$

La primera operación se ilustra en la figura 6.13. El ejemplo que dimos antes para ilustrar la operación de PRODUCTO CARTESIANO lo podemos especificar, con la operación REUNIÓN, sustituyendo las dos operaciones

$$\text{DEPENDIENTES\_EMP} \leftarrow \text{NOMBRESEMP X DEPENDIENTE}$$
$$\text{DEPENDIENTES\_REALES} \leftarrow \sigma_{\text{NSS=NSSE}}(\text{DEPENDIENTES\_EMP})$$

por

$$\text{DEPENDIENTES\_REALES} \leftarrow \text{NOMBRESEMP} \bowtie_{\text{NSS=NSSE}} \text{DEPENDIENTE}$$

La forma general de una operación REUNIÓN con dos relaciones $R(A_1, A_2, ..., A_n)$ y $S(B_1, B_2, ..., B_m)$ es

$$R \bowtie_{\text{<condición de reunión>}} S$$

El resultado de la REUNIÓN es una relación Q con $n + m$ atributos $Q(A_1, A_2, ..., A_n, B_1, B_2, ..., B_m)$, en ese orden; Q tiene una tupla por cada combinación de tuplas —una de R y una de S— *siempre que la combinación satisfaga la condición de reunión*. Ésta es la principal diferencia entre el PRODUCTO CARTESIANO y la REUNIÓN: en la REUNIÓN, sólo aparecen en el resultado combinaciones de tuplas que satisfagan la condición de reunión; en cambio, en el PRODUCTO CARTESIANO, *todas* las combinaciones de tuplas se incluyen en el resultado. La condición de reunión se especifica en términos de los atributos de las dos relaciones, R y S, y se evalúa para cada combinación de tuplas. Cada combinación de tuplas para la cual la evaluación de la condición con los valores de los atributos produzca el resultado "verdadero" se incluirá en la relación resultante, Q, como una sola tupla.

Una condición de reunión tiene la forma:

<condición> Y <condición> Y ··· Y <condición>

donde cada condición tiene la forma $A_i\ \theta\ B_j$, A_i es un atributo de R, B_j es un atributo de S, A_i y B_j tienen el mismo dominio y θ es uno de los operadores de comparación $\{=, <, \leq, >, \geq, \neq\}$. Una operación de REUNIÓN con una condición general de reunión como ésta se denomina **REUNIÓN THETA**. Las tuplas cuyos atributos de reunión sean nulos *no* aparecen en el resultado.

GTE_DEPTO	NOMBRED	NÚMEROD	NSSGTE	• • •	NOMBREP	INIC	APELLIDO	NSS	• • •
	Investigación	5	333445555	• • •	Federico	T	Vizcarra	333445555	• • •
	Administración	4	987654321	• • •	Jazmín	S	Valdés	987654321	• • •
	Dirección	1	888665555	• • •	Jaime	E	Botello	888665555	• • •

Figura 6.13 La operación REUNIÓN.

La REUNIÓN más común implica condiciones de reunión con comparaciones de igualdad exclusivamente. Una REUNIÓN así, en la que el único operador de comparación empleado es =, se llama **EQUIRREUNIÓN**. Los dos ejemplos que hemos considerado han sido EQUIRREUNIONES. Observe que en el resultado de una EQUIRREUNIÓN siempre tenemos uno o más pares de atributos con *valores idénticos* en todas las tuplas. Por ejemplo, en la figura 6.13, los valores de los atributos NSSGTE y NSS son idénticos en todas las tuplas de GTE_DEPTO porque se especificó la condición de reunión de igualdad para estos dos atributos. Puesto que uno de cada par de atributos con valores idénticos es superfluo, se ha creado una nueva operación, llamada **REUNIÓN NATURAL**, para deshacerse del segundo atributo en una condición de equirreunión. Denotamos la reunión natural con *. Se trata básicamente de una equirreunión seguida de la eliminación de los atributos superfluos. La definición estándar de la REUNIÓN NATURAL exige que los dos atributos de reunión (o cada par de atributos de reunión) tengan el mismo nombre. Si no es así, primero se aplica una operación de cambio de nombre. Un ejemplo es

$$\text{DEPTO (NOMBRED, NÚMD, NSSGTE, FECHAINICGTE)} \leftarrow \text{DEPARTAMENTO}$$
$$\text{DEPTO\_PROY} \leftarrow \text{PROYECTO} * \text{DEPTO}$$

El atributo NÚMD se denomina **atributo de reunión**. La relación resultante se ilustra en la figura 6.14(a). En la relación DEPTO_PROY, cada tupla combina una tupla PROYECTO con la tupla DEPARTAMENTO del departamento que controla el proyecto. En la relación resultante sólo se conserva un *atributo de reunión*.

Si los atributos sobre los cuales se especifica la reunión natural *tienen los mismos nombres en las dos relaciones*, no hace falta el cambio de nombres. Para aplicar una reunión natural sobre el atributo NÚMEROD de DEPARTAMENTO y LUGARES_DEPTOS basta con escribir

$$\text{LUG\_DEPTOS} \leftarrow \text{DEPARTAMENTO} * \text{LUGARES\_DEPTOS}$$

La relación resultante se muestra en la figura 6.14(b), que combina cada departamento con sus lugares y tiene una tupla por cada lugar. En general, la REUNIÓN NATURAL se realiza igualando *todos* los pares de atributos que tienen el mismo nombre en las dos relaciones. Puede haber una lista de atributos de reunión de cada relación, y cada par corespondiente debe tener el mismo nombre.

(a)

DEPTO_PROY	NOMBREP	NÚMEROP	LUGARP	NÚMD	NOMBRED	NSSGTE	FECHAINICGTE
	ProductoX	1	Belén	5	Investigación	333445555	22-MAY-78
	ProductoY	2	Sacramento	5	Investigación	333445555	22-MAY-78
	ProductoZ	3	Higueras	5	Investigación	333445555	22-MAY-78
	Automatización	10	Santiago	4	Administración	987654321	01-ENE-85
	Reorganización	20	Higueras	1	Dirección	888665555	19-JUN-71
	Nuevasprestaciones	30	Santiago	4	Administración	987654321	01-ENE-85

(b)

LUG_DEPTOS	NOMBRED	NÚMEROD	NSSGTE	FECHAINICGTE	LUGARD
	Dirección	1	888665555	19-JUN-71	Higueras
	Administración	4	987654321	01-ENE-85	Santiago
	Investigación	5	333445555	22-MAY-78	Belén
	Investigación	5	333445555	22-MAY-78	Sacramento
	Investigación	5	333445555	22-MAY-78	Higueras

Figura 6.14 La operación REUNIÓN NATURAL. (a) DEPTO_PROY ← PROYECTO * DEPTO. (b) LUG_ DEPTOS ← DEPARTAMENTO * LUGARES_DEPTOS.

Una definición más general de la REUNIÓN NATURAL es

$$Q \leftarrow R *_{(<lista1>),(<lista2>)} S$$

En este caso, <lista1> especifica una lista de i atributos de R, y <lista2> especifica una lista de i atributos de S. Las listas sirven para formar condiciones de comparación de igualdad entre pares de atributos correspondientes; condiciones que después se eslabonan con el operador Y. Sólo la lista que corresponde a los atributos de la primera relación R —<lista1>— se conserva en el resultado Q.

Cabe señalar que, si ninguna combinación de tuplas satisface la condición de reunión, el resultado de una REUNIÓN es una relación vacía con cero tuplas. En general, si R tiene n_R tuplas y S tiene n_S tuplas, el resultado de una operación de REUNIÓN $R \bowtie_{<condición de reunión>} S$ tendrá entre cero y $n_R * n_S$ tuplas. El tamaño esperado del resultado de la reunión dividido entre el tamaño máximo $n_R * n_S$ da lugar a una razón llamada **selectividad de reunión**, que es una propiedad de cada condición de reunión. Si no hay <condición de reunión> que satisfacer, todas las combinaciones de tuplas califican y la REUNIÓN se convierte en un PRODUCTO CARTESIANO, llamado también REUNIÓN CRUZADA.

6.5.6 *Conjunto completo de operaciones del álgebra relacional*

Se ha demostrado que el conjunto de operaciones del álgebra relacional $\{\sigma, \pi, \cup, -, X\}$ es un conjunto **completo**; es decir, cualquiera de las otras operaciones del álgebra relacional se puede expresar como una *secuencia de operaciones de este conjunto*. Por ejemplo, la operación INTERSECCIÓN se puede expresar empleando UNIÓN y DIFERENCIA como sigue:

$$R \cap S \equiv (R \cup S) - ((R - S) \cup (S - R))$$

Aunque, en términos estrictos, la INTERSECCIÓN no es indispensable, resulta poco cómodo especificar esta expresión compleja cada vez que deseemos especificar una intersección. Como ejemplo adicional, una operación de REUNIÓN se puede especificar como un PRODUCTO CARTESIANO seguido de una operación SELECCIONAR, como explicamos antes:

$$R \bowtie_{<condición>} S \equiv \sigma_{<condición>} (R \times S)$$

De manera similar, una REUNIÓN NATURAL se puede especificar como un PRODUCTO CARTESIANO seguido de operaciones SELECCIONAR y PROYECTAR. Así pues, las diversas operaciones de REUNIÓN *tampoco son estrictamente necesarias* para el poder expresivo del álgebra relacional; sin embargo, son muy importantes —lo mismo que la operación INTERSECCIÓN— porque son cómodas y se emplean con mucha frecuencia en las aplicaciones de bases de datos. Otras operaciones se han incluido en el álgebra relacional por comodidad más que por necesidad. Analizaremos una de ellas, la operación DIVISIÓN, en la siguiente sección.

6.5.7 *La operación DIVISIÓN*★

La operación DIVISIÓN es útil para un tipo especial de consultas que se presenta ocasionalmente en aplicaciones de bases de datos. Un ejemplo es: "obtener los nombres de los empleados que trabajan en *todos* los proyectos en los que trabaja 'José Silva'". Para expresar esta consulta con la operación DIVISIÓN, procedemos como sigue. Primero, obtenemos la lista de números de los proyectos en los que trabaja 'José Silva', colocando el resultado en la relación intermedia NÚMSP_SILVA:

$$\text{SILVA} \leftarrow \sigma_{\text{NOMBREP='José' Y APELLIDO='Silva'}}(\text{EMPLEADO})$$
$$\text{NÚMSP\_SILVA} \leftarrow \pi_{\text{NÚMP}}(\text{TRABAJA\_EN} *_{\text{NSSE=NSS}} \text{SILVA})$$

En seguida, creamos una relación intermedia NSS_NÚMSP que incluye una tupla <NÚMP, NSSE> por cada vez que el empleado cuyo número de seguro social es NSSE trabaja en el proyecto cuyo número es NÚMP:

$$\text{NSS\_NÚMSP} \leftarrow \pi_{\text{NÚMP, NSSE}}(\text{TRABAJA\_EN})$$

Por último, aplicamos la operación DIVISIÓN a las dos relaciones, obteniendo los números de seguro social de los empleados que queremos:

$$\text{NSSS(NSS)} \leftarrow \text{NSS\_NÚMSP} \div \text{NÚMSP\_SILVA}$$
$$\text{RESULTADO} \leftarrow \pi_{\text{NOMBREP, APELLIDO}}(\text{NSSS} * \text{EMPLEADO})$$

Las operaciones anteriores se muestran en la figura 6.15(a). En general, la operación DIVISIÓN se aplica a dos relaciones $R(Z) \div S(X)$, donde $X \subseteq Z$. Sea $Y = Z - X$; es decir, sea Y el conjunto de atributos de R que no son atributos de S. El resultado de DIVISIÓN es una relación $T(Y)$ que incluye una tupla t si una tupla t_R cuyo $t_R[Y] = t$ aparece en R, con $t_R[X] = t_S$ *para cada tupla* t_S en S. Esto significa que, para que una tupla t aparezca en el resultado T de la DIVISIÓN, los valores de t deben aparecer en R en combinación con *todas* las tuplas de S.

La figura 6.15(b) ilustra un operador DIVISIÓN donde tanto $X = \{A\}$ como $Y = \{B\}$ son atributos individuales. Observe que b_1 y b_4 aparecen en R en combinación con las tres tuplas de S; es por ello que aparecen en la relación resultante T. Todos los demás valores de B en R no aparecen con todas las tuplas de S y no se seleccionan: b_2 no aparece con a_2 y b_3 no aparece con a_1.

El operador DIVISIÓN se puede expresar como una secuencia de operaciones π, X y –, como sigue:

$$T_1 \leftarrow \pi_Y(R)$$
$$T_2 \leftarrow \pi_Y((S \times T_1) - R)$$
$$T \leftarrow T_1 - T_2$$

6.6 Otras operaciones relacionales★

Algunas solicitudes que se hacen comúnmente a las bases de datos no se pueden atender con las operaciones estándar del álgebra relacional descritas en la sección 6.5. La mayoría de los lenguajes de consulta comerciales para los SGBD relacionales cuentan con mecanismos para atender dichas solicitudes, y en esta sección definiremos algunas operaciones adicionales para expresar las consultas. Estas operaciones amplían el poder expresivo del álgebra relacional.

6.6.1 *Funciones agregadas*

El primer tipo de solicitud que no se puede expresar en el álgebra relacional es la especificación de **funciones agregadas** matemáticas sobre colecciones de valores de la base de datos. Un ejemplo sería la obtención del salario medio o total de todos los empleados o el número de tuplas de empleados. Entre las funciones que suelen aplicarse a colecciones de valores numéricos están SUMA, PROMEDIO, MÁXIMO y MÍNIMO. La función CUENTA sirve para contar tuplas. Todas estas funciones pueden aplicarse a una colección de tuplas.

(a)

NSS_NÚMSP	NÚMP	NSSE
	1	123456789
	2	123456789
	3	666884444
	1	453453453
	2	453453453
	2	333445555
	3	333445555
	10	333445555
	20	333445555
	30	999887777
	10	999887777
	10	987987987
	30	987987987
	30	987654321
	20	987654321
	20	888665555

NÚMSP_SILVA	NÚMP
	1
	2

NSSS	NSS
	123456789
	453453453

(b)

R	A	B
	a_1	b_1
	a_2	b_1
	a_3	b_1
	a_4	b_1
	a_1	b_2
	a_3	b_2
	a_2	b_3
	a_3	b_3
	a_4	b_3
	a_1	b_4
	a_2	b_4
	a_3	b_4

S	A
	a_1
	a_2
	a_3

T	B
	b_1
	b_4

Figura 6.15 La operación DIVISIÓN. (a) División de NSS_NÚMSP entre NÚMSP_SILVA. (b) $T \leftarrow R \div S$.

Otro tipo de solicitud que se hace con frecuencia implica agrupar las tuplas de una relación según el valor de algunos de sus atributos y después aplicar una función agregada independientemente a cada grupo. Un ejemplo sería agrupar las tuplas de empleados por ND,

de modo que cada grupo incluya las tuplas de los empleados que trabajan en el mismo departamento. Después podríamos preparar una lista de valores de ND junto con, digamos, el salario medio de los empleados del departamento.

Podemos definir una operación FUNCIÓN† empleando el símbolo $\mathfrak{F}$ (pronunciado "F gótica") para especificar estos tipos de solicitudes, como sigue:

$$<\text{atributos de agrupación}>\ \mathfrak{F}_{<\text{lista de funciones}>}\ (<\text{nombre de la relación}>)$$

donde <atributos de agrupación> es una lista de atributos de la relación especificada en <nombre de la relación>, y <lista de funciones> es una lista de pares (<función> <atributo>). En cada uno de esos pares, <función> es una de las funciones permitidas —como SUMA, PROMEDIO, MÁXIMO, MÍNIMO, CUENTA— y <atributo> es un atributo de la relación especificada en <nombre de la relación>. La relación resultante tiene los atributos de agrupación más un atributo por cada elemento de la lista de funciones. Por ejemplo, para obtener los números de departamento, el número de empleados de cada departamento y su salario medio, escribimos

$$\text{R(ND, NÚM\_DE\_EMPLEADOS, SAL\_MEDIO)} \leftarrow {}_{\text{ND}}\mathfrak{F}_{\text{CUENTA NSS, PROMEDIO SALARIO}}\ (\text{EMPLEADO})$$

El resultado de esta operación se muestra en la figura 6.16(a).

En el ejemplo anterior, especificamos una lista de nombres de atributos (entre paréntesis) para la relación resultante R. Si no se especifica una lista así, cada uno de los atributos de la relación resultante que correspondan a la lista de funciones tendrán como nombre la concatenación del nombre de la función y el nombre del atributo al que se aplica la función, en la forma <función>_<atributo>. Por ejemplo, en la figura 6.16(b) se muestra el resultado de la siguiente operación:

$${}_{\text{ND}}\mathfrak{F}_{\text{CUENTA NSS, PROMEDIO SALARIO}}\ (\text{EMPLEADO})$$

(a)

R	ND	NÚM_DE_EMPLEADOS	SAL_MEDIO
	5	4	33250
	4	3	31000
	1	1	55000

(b)

ND	CUENTA_NSS	PROMEDIO_SALARIO
5	4	33250
4	3	31000
1	1	55000

(c)

CUENTA_NSS	PROMEDIO_SALARIO
8	35125

Figura 6.16 La operación FUNCIÓN. (a) $R(\text{ND, NÚM\_DE\_EMPLEADOS, SAL\_MEDIO} \leftarrow {}_{\text{ND}}\mathfrak{F}_{\text{CUENTA NSS, PROMEDIO SALARIO}}\ (\text{EMPLEADO})$. (b) ${}_{\text{ND}}\mathfrak{F}_{\text{CUENTA NSS, PROMEDIO SALARIO}}\ (\text{EMPLEADO})$. (c) $\mathfrak{F}_{\text{CUENTA NSS, PROMEDIO SALARIO}}\ (\text{EMPLEADO})$.

†No existe una única notación convenida para especificar funciones agregadas.

Si no se especifican atributos de agrupación, las funciones se aplicarán a los valores de los atributos de *todas las tuplas* de la relación, y la relación resultante tendrá *una sola tupla*. Por ejemplo, en la figura 6.16(c) se muestra el resultado de la siguiente operación:

$$\mathfrak{F}_{\text{CUENTA NSS, PROMEDIO SALARIO}}\ (\text{EMPLEADO})$$

Vale la pena subrayar que el resultado de aplicar una función agregada es una relación, no un número escalar, aunque tenga un solo valor.

6.6.2 *Operaciones de cerradura recursiva*

Otro tipo de operación que, en general, no puede especificarse en el álgebra relacional es la **cerradura recursiva** (a veces llamada **cierre recursivo**). Esta operación se aplica a un **vínculo recursivo** entre tuplas del mismo tipo, como el vínculo entre un empleado y un supervisor. Este vínculo se describe mediante la clave externa NSSSUPER de la relación EMPLEADO en las figuras 6.6 y 6.7, pues relaciona cada tupla de empleado (en el papel de supervisado) con otra tupla de empleado (en el papel de supervisor). Un ejemplo de operación recursiva sería obtener todos los supervisados de un empleado *e* en todos los niveles; esto es, todos los empleados *e'* supervisados directamente por *e*, todos los empleados *e''* supervisados directamente por un empleado *e'*, todos los empleados *e'''* supervisados directamente por un empleado *e''*, y así sucesivamente. Aunque en el álgebra relacional resulta sencillo especificar todos los empleados supervisados por *e en un nivel específico*, no lo es especificar todos los supervisados en *todos* los niveles. Por ejemplo, para especificar los NSS de todos los empleados *e'* supervisados directamente —*en el nivel 1*— por el empleado *e* cuyo nombre es 'Jaime Botello' (véase la Fig. 6.6), podemos aplicar las siguientes operaciones:

$$\text{NSS\_BOTELLO} \leftarrow \pi_{\text{NSS}}(\sigma_{\text{NOMBREP='Jaime' Y APELLIDO='Botello'}}(\text{EMPLEADO}))$$
$$\text{SUPERVISIÓN(NSS1, NSS2)} \leftarrow \pi_{\text{NSS, NSSSUPER}}(\text{EMPLEADO})$$
$$\text{RESULTADO1(NSS)} \leftarrow \pi_{\text{NSS1}}(\text{SUPERVISIÓN} \bowtie_{\text{NSS2=NSS}} \text{NSS\_BOTELLO})$$

Para obtener todos los empleados supervisados por Botello en el nivel 2 —esto es, todos los empleados *e''* supervisados por algún empleado *e'* que es supervisado directamente por Botello— podemos aplicar otra REUNIÓN al resultado de la primera consulta, así:

$$\text{RESULTADO2(NSS)} \leftarrow \pi_{\text{NSS1}}(\text{SUPERVISIÓN} \bowtie_{\text{NSS2=NSS}} \text{RESULTADO1})$$

Si queremos obtener los dos conjuntos de empleados supervisados en los niveles 1 y 2 por 'Jaime Botello', podemos aplicar la operación UNIÓN a los dos resultados, así:

$$\text{RESULTADO3} \leftarrow (\text{RESULTADO1} \cup \text{RESULTADO2})$$

Los resultados de estas consultas se ilustran en la figura 6.17. Aunque es posible obtener empleados en cada uno de los niveles y luego efectuar su UNIÓN, no podemos, en general, especificar una consulta como "obtener los supervisados de 'Jaime Botello' en todos los niveles" si no conocemos el *número máximo de niveles*, porque necesitaríamos un mecanismo de ciclo.

6.6.3 *Operaciones de REUNIÓN EXTERNA y UNIÓN EXTERNA*

Por último, analizaremos algunas extensiones de las operaciones REUNIÓN y UNIÓN. Las operaciones de REUNIÓN antes descritas seleccionan tuplas que satisfacen la condición de reunión. Por ejemplo, con una operación de REUNIÓN NATURAL $R * S$, sólo las tuplas de R que tienen tuplas coincidentes en S —y viceversa— aparecen en el resultado. Por tanto, las tuplas sin una

(El NSS de Botello es 888665555)

	(NSS)	(NSSSUPER)
SUPERVISIÓN	NSS1	NSS2
	123456789	333445555
	333445555	888665555
	999887777	987654321
	987654321	888665555
	666884444	333445555
	453453453	333445555
	987987987	987654321
	888665555	nulo

RESULTADO 1	NSS
	333445555
	987654321

(Supervisados por Botello)

RESULTADO 2	NSS
	123456789
	999887777
	666884444
	453453453
	987987987

(Supervisados por los subordinados de Botello)

RESULTADO 3	NSS
	123456789
	999887777
	666884444
	453453453
	987987987
	333445555
	987654321

(RESULTADO1 ∪ RESULTADO2)

Figura 6.17 Recursión de dos niveles.

"tupla relacionada" se eliminan del resultado, y lo mismo sucede con las tuplas que tienen nulo en los atributos de reunión. Podemos usar un conjunto de operaciones, llamadas REUNIONES EXTERNAS, cuando queremos conservar en el resultado todas las tuplas que están en R o en S, o en ambas, ya sea que tengan o no tuplas coincidentes en la otra relación.

Por ejemplo, suponga que deseamos una lista de todos los nombres de empleados y también el nombre de los departamentos que dirigen, *si es el caso que dirijan un departamento*; podemos aplicar una operación **REUNIÓN EXTERNA IZQUIERDA**, denotado por $\rtimes$, para obtener el resultado como sigue:

$$\text{TEMP} \leftarrow (\text{EMPLEADO} \rtimes_{\text{NSS = NSSGTE}} \text{DEPARTAMENTO})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{NOMBREP, INIC, APELLIDO, NOMBRED}}\ (\text{TEMP})$$

La operación de REUNIÓN EXTERNA IZQUIERDA conserva en $R \rtimes S$ todas las tuplas de la *primera* relación R (o relación de la izquierda); si no se encuentra una tupla coincidente en S, los atributos de S del resultado se "rellenan" con valores nulos. El resultado de estas operaciones se muestra en la figura 6.18.

Una operación similar, la REUNIÓN EXTERNA DERECHA, denotada por $\ltimes$, conserva en el resultado de $R \ltimes S$ todas las tuplas de la *segunda* relación S (la de la derecha). Una tercera operación, la REUNIÓN EXTERNA COMPLETA denotada por $\rtimes\ltimes$, conserva todas las tuplas de ambas relaciones, izquierda y derecha, cuando no se encuentran tuplas coincidentes, rellenándolas con valores nulos si es necesario.

La operación de **UNIÓN EXTERNA** se creó para efectuar la unión de tuplas de dos relaciones que *no son compatibles con la unión*. Esta operación efectuará la UNIÓN de tuplas de

RESULTADO	NOMBREP	INIC	APELLIDO	NOMBRED
	José	B	Silva	nulo
	Federico	T	Vizcarra	Investigación
	Alicia	J	Zapata	nulo
	Jazmín	S	Valdés	Administración
	Ramón	K	Nieto	nulo
	Josefa	A	Esparza	nulo
	Ahmed	V	Jabbar	nulo
	Jaime	E	Botello	Dirección

Figura 6.18 La operación REUNIÓN EXTERNA IZQUIERDA.

dos relaciones que sean **parcialmente compatibles**, lo que significa que sólo algunos de sus atributos son compatibles con la unión. En el resultado se conservan los atributos no compatibles de cualquiera de las relaciones, y las tuplas que no tienen valores para dichos atributos se rellenan con valores nulos. Por ejemplo, se puede aplicar una UNIÓN EXTERNA a dos relaciones cuyos esquemas son ESTUDIANTE (Nombre, NSS, Departamento, Asesor) y PROFESORADO (Nombre, NSS, Departamento, Rango). El esquema de la relación resultante es R(Nombre, NSS, Departamento, Asesor, Rango), y todas las tuplas de ambas relaciones se incluyen en el resultado. Las tuplas de estudiantes tendrán nulos en el atributo Rango, y las tuplas de profesorado tendrán nulos en el atributo Asesor. Una tupla que exista en ambas relaciones tendrá valores para todos sus atributos.

Otra capacidad que tienen casi todos los lenguajes comerciales (pero no el álgebra relacional) es la de especificar operaciones con los valores después de extraerlos de la base de datos. Por ejemplo, se puede aplicar operaciones aritméticas como +, – y * a los valores numéricos.

6.7 Ejemplos de consultas en el álgebra relacional

Ahora presentaremos ejemplos adicionales para ilustrar el empleo de las operaciones del álgebra relacional; todos ellos se refieren a la base de datos de la figura 6.6. En general, la misma consulta puede expresarse de muchas formas mediante las diversas operaciones. Expresaremos cada una de las consultas de una manera y dejaremos que el lector proponga formulaciones equivalentes.

CONSULTA 1
Obtener el nombre y la dirección de todos los empleados que trabajan para el departamento 'Investigación'.

$$\text{DEPTO\_INVEST} \leftarrow \sigma_{\text{NOMBRED='Investigación'}}(\text{DEPARTAMENTO})$$
$$\text{EMPS\_DEPTO\_INVEST} \leftarrow (\text{DEPTO\_INVEST} \bowtie_{\text{NÚMEROD=ND}} \text{EMPLEADO})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{NOMBREP, APELLIDO, DIRECCIÓN}}(\text{EMPS\_DEPTO\_INVEST})$$

Esta consulta se podría especificar de otras maneras; por ejemplo, se podría invertir el orden de las operaciones REUNIÓN y SELECCIONAR, o se podría sustituir la REUNIÓN por una REUNIÓN NATURAL.

CONSULTA 2
Para cada proyecto ubicado en 'Santiago', obtener una lista con el número de proyecto, el número del departamento que lo controla, y el apellido, la dirección y la fecha de nacimiento del gerente de dicho departamento.

$$\text{PROYS\_SANTIAGO} \leftarrow \sigma_{\text{LUGARP='Santiago'}}(\text{PROYECTO})$$
$$\text{DEPTO\_CONTR} \leftarrow (\text{PROYS\_SANTIAGO} \bowtie_{\text{NÚMD=NÚMEROD}} \text{DEPARTAMENTO})$$
$$\text{GTE\_DEPTO\_PROY} \leftarrow (\text{DEPTO\_CONTR} \bowtie_{\text{NSSGTE=NSS}} \text{EMPLEADO})$$
$$\text{RESULTADO} \leftarrow \pi_{\text{NÚMEROP, NÚMD, APELLIDO, DIRECCIÓN, FECHAN}}(\text{GTE\_DEPTO\_PROY})$$

CONSULTA 3
Buscar los nombres de los empleados que trabajan en *todos* los proyectos controlados por el departamento número 5.

$$\text{PROYS\_DEPTO5(NÚMP)} \leftarrow \pi_{\text{NÚMEROP}}(s_{\text{NÚMD=5}}(\text{PROYECTO}))$$
$$\text{EMP\_PROY(NSS, NÚMP)} \leftarrow \pi_{\text{NSSE, NÚMP}}(\text{TRABAJA\_EN})$$
$$\text{RESUL\_NSSS\_EMP} \leftarrow \text{EMP\_PROY} \div \text{PROYS\_DEPTO5}$$
$$\text{RESULTADO} \leftarrow \pi_{\text{APELLIDO, NOMBREP}}(\text{RESUL\_NSSS\_EMP} * \text{EMPLEADO})$$

CONSULTA 4
Preparar una lista con los números de los proyectos en que interviene un empleado cuyo apellido es 'Silva', ya sea como trabajador o como gerente del departamento que controla el proyecto.

$$\text{SILVAS(NSSE)} \leftarrow \pi_{\text{NSS}}(\sigma_{\text{APELLIDO='Silva'}}(\text{EMPLEADO}))$$
$$\text{SILVA\_TRAB\_PROYS} \leftarrow \pi_{\text{NÚMP}}(\text{TRABAJA\_EN} * \text{SILVAS})$$
$$\text{GTES} \leftarrow \pi_{\text{APELLIDO, NÚMEROD}}(\text{EMPLEADO} \bowtie_{\text{NSS=NSSGTE}} \text{DEPARTAMENTO})$$
$$\text{SILVA\_GTES} \leftarrow \sigma_{\text{APELLIDO='Silva'}}(\text{GTES})$$
$$\text{DEPTOS\_DIRIG\_SILVA(NÚMD)} \leftarrow \pi_{\text{NÚMEROD}}(\text{SILVA\_GTES})$$
$$\text{SILVA\_GTE\_PROYS(NÚMP)} \leftarrow \pi_{\text{NÚMEROP}}(\text{DEPTOS\_DIRIG\_SILVA} * \text{PROYECTO})$$
$$\text{RESULTADO} \leftarrow (\text{SILVA\_TRAB\_PROYS} \cup \text{SILVA\_GTE\_PROYS})$$

CONSULTA 5
Prepare una lista con los nombres de todos los empleados que tienen dos o más dependientes.

En términos estrictos, esta consulta *no puede realizarse en el álgebra relacional*. Tenemos que usar la operación FUNCIÓN con la función agregada CUENTA. Suponemos que los dependientes del *mismo* empleado tienen valores *distintos* de NOMBRE_DEPENDIENTE.

$$T_1(\text{NSS, NÚM\_DE\_DEPS}) \leftarrow {}_{\text{NSSE}}\Im_{\text{CUENTA NOMBRE\_DEPENDIENTE}}(\text{DEPENDIENTE})$$
$$T_2 \leftarrow \sigma_{\text{NÚM\_DE\_DEPS} \geq 2}(T_1)$$
$$\text{RESULTADO} \leftarrow \pi_{\text{APELLIDO, NOMBREP}}(T_2 * \text{EMPLEADO})$$

CONSULTA 6
Obtener los nombres de los empleados que no tienen dependientes.

$$\text{TODOS\_EMPS} \leftarrow \pi_{\text{NSS}}(\text{EMPLEADO})$$
$$\text{EMPS\_CON\_DEPS(NSS)} \leftarrow \pi_{\text{NSSE}}(\text{DEPENDIENTE})$$

EMPS_SIN_DEPS ← (TODOS_EMPS - EMPS_CON_DEPS)

RESULTADO ← $\pi_{\text{APELLIDO, NOMBREP}}$(EMPS_SIN_DEPS * EMPLEADO)

CONSULTA 7

Obtener los nombres de los gerentes que tienen por lo menos un dependiente.

GTES(NSS) ← π_{NSSGTE}(DEPARTAMENTO)

EMPS_CON_DEPS(NSS) ← π_{NSSE}(DEPENDIENTE)

GTES_CON_DEPS ← (GTES ∩ EMPS_CON_DEPS)

RESULTADO ← $\pi_{\text{APELLIDO, NOMBREP}}$(GTES_CON_DEPS * EMPLEADO)

Como dijimos antes, en general podemos especificar la misma consulta de varias formas distintas. Por ejemplo, es frecuente que las operaciones se puedan aplicar en diversas secuencias. Por añadidura, algunas operaciones se pueden sustituir por otras; por ejemplo, la operación INTERSECCIÓN de la consulta 7 se puede reemplazar por una reunión natural. Como ejercicio, trate de efectuar los ejemplos de consultas dados aquí con operaciones distintas. En los capítulos 7 y 8 mostraremos cómo se pueden expresar estas consultas en otros lenguajes relacionales.

6.8 Uso de la transformación ER-relacional para el diseño de bases de datos relacionales*

En esta sección explicaremos la forma de derivar un esquema de base de datos relacional a partir de un esquema conceptual creado empleando el modelo de entidad-vínculo(ER) (véase el Cap. 3). Muchas herramientas CASE (*computer-aided software engineering*: ingeniería de software asistida por computador) se basan en el modelo ER y en sus variaciones. Los diseñadores de bases de datos utilizan interactivamente estas herramientas computarizadas para crear un esquema ER para su aplicación de base de datos. Muchas herramientas se valen de diagramas ER o variaciones de ellos para crear el esquema gráficamente, y luego lo convierten automáticamente en un esquema de base de datos relacional expresado en el DDL de un SGBD relacional específico.

En la sección 6.8.1 bosquejaremos un algoritmo que puede convertir un esquema ER en el esquema de base de datos relacional correspondiente. Después, en la sección 6.8.2, presentaremos un resumen de las correspondencias entre los elementos del modelo ER y los del modelo relacional.

6.8.1 *Algoritmo de transformación ER-modelo relacional*

Ahora describiremos informalmente los pasos de un algoritmo para la transformación ER-modelo relacional.

El esquema relacional COMPAÑÍA de la figura 6.5 se puede derivar del esquema ER de la figura 3.2 siguiendo estos pasos. Ilustraremos cada paso con ejemplos del esquema COMPAÑÍA.

PASO 1: Por cada tipo normal de entidades *E* del esquema ER, se crea una relación *R* que contenga todos los atributos simples de *E*. Se incluyen sólo los atributos simples componentes de un atributo compuesto. Se elige uno de los atributos clave de E como clave primaria de *R*. Si la clave elegida es compuesta, el conjunto de atributos simples que la forman constituirá la clave primaria de *R*.

En nuestro ejemplo, creamos las relaciones EMPLEADO, DEPARTAMENTO y PROYECTO de la figura 6.5, que corresponden a los tipos de entidades normales EMPLEADO, DEPARTAMENTO y PROYECTO de la figura 3.2. A éstas en ocasiones se les denomina relaciones "entidades". No se incluyen todavía los atributos de clave externa y de vínculo; se agregarán durante los pasos subsecuentes. Entre ellos están los atributos NSSSUPER y ND de EMPLEADO; NSSGTE y FECHAINICGTE de DEPARTAMENTO, y NÚMD de PROYECTO. Escogemos NSS, NÚMEROD y NÚMEROP como claves primarias de las relaciones EMPLEADO, DEPARTAMENTO y PROYECTO, respectivamente.

PASO 2: Por cada tipo de entidad débil *D* del esquema ER con tipo de entidades propietarias *E*, se crea una relación *R* y se incluyen todos los atributos simples (o componentes simples de los atributos compuestos) de *D* como atributos de *R*. Además, se incluyen como atributos de clave externa de *R* los atributos de clave primaria de la relación o relaciones que corresponden al tipo o tipos de entidades propietarias; con esto damos cuenta del tipo de vínculo identificador de *D*. La clave primaria de *R* es la combinación de las claves primarias de las propietarias y la clave parcial de *D*, si existe.

En nuestro ejemplo, creamos la relación DEPENDIENTE en este paso, que corresponde al tipo de entidades débil DEPENDIENTE. Incluimos la clave primaria de la relación EMPLEADO —que corresponde al tipo de entidades propietarias— como atributo de clave externa de DEPENDIENTE; cambiamos su nombre a NSSE, aunque no era preciso hacerlo. La clave primaria de la relación DEPENDIENTE es la combinación {NSSE, NOMBRE_DEPENDIENTE} porque NOMBRE_DEPENDIENTE es la clave parcial de DEPENDIENTE.

PASO 3: Por cada tipo de vínculo binario 1:1 *R* del esquema ER, se identifican las relaciones *S* y *T* que corresponden a los tipos de entidades que participan en *R*. Se escoge una de las relaciones —digamos *S*— y se incluye como clave externa en *S* la clave primaria de *T*. Es mejor elegir un tipo de entidades con participación total en *R* en el papel de *S*. Se incluyen todos los atributos simples (o componentes simples de los atributos compuestos) del tipo de vínculos 1:1 *R* como atributos de *S*.

En nuestro ejemplo, transformaremos el tipo de vínculo 1:1 DIRIGE de la figura 3.2 eligiendo DEPARTAMENTO para desempeñar el papel de *S*, debido a que su participación en DIRIGE es total (todo departamento tiene un gerente). Incluimos la clave primaria de la relación EMPLEADO como clave externa en la relación DEPARTAMENTO y cambiamos su nombre a NSSGTE. También incluimos el atributo simple FechaInic de DIRIGE en la relación DEPARTAMENTO y cambiamos su nombre a FECHAINICGTE.

Cabe señalar que puede establecerse una transformación alternativa de un tipo de vínculos 1:1 si combinamos los dos tipos de entidades y el vínculo en una sola relación. Esto resulta apropiado sobre todo cuando las dos participaciones son totales y cuando los tipos de entidades no participan en ningún otro tipo de vínculos.

PASO 4: Por cada tipo de vínculos normal (no débil) binario 1:*N* *R*, se identifica la relación *S* que representa el tipo de entidades participante del *lado N* del tipo de vínculos. Se incluye como clave externa en *S* la clave primaria de la relación *T* que representa al otro tipo de entidades que participa en *R*; la razón es que cada ejemplar de entidad del lado *N* está relacionado con un máximo de un ejemplar de entidad del lado 1. Se incluyen todos los atributos

simples (o componentes simples de los atributos compuestos) del tipo de vínculos 1:*N* como atributos de *S*.

En nuestro ejemplo, establecemos ahora la transformación de los tipos de vínculos PERTENECE_A, CONTROLA y SUPERVISIÓN de la figura 3.2. En el caso de PERTENECE_A incluimos la clave primaria de la relación DEPARTAMENTO como clave externa en la relación EMPLEADO y la llamamos ND. En el caso de SUPERVISIÓN, incluimos la clave primaria de la relación EMPLEADO como clave externa en la relación EMPLEADO misma y la llamamos NSSSUPER. El vínculo CONTROLA corresponde al atributo de clave externa NÚMD de PROYECTO.

PASO 5: Por cada tipo de vínculos binario M:*N* *R*, se crea una nueva relación *S* para representar *R*. Se incluyen como atributos de clave externa en *S* las claves primarias de las relaciones que representan los tipos de entidades participantes; su combinación constituirá la clave primaria de *S*. También se incluyen todos los atributos simples (o componentes simples de los atributos compuestos) del tipo de vínculos M:*N* como atributos de *S*. Observe que no podemos representar un tipo de vínculos M:*N* con un solo atributo de clave externa en una de las relaciones participantes —como hicimos en el caso de los tipos de vínculos 1:1 y 1:*N*— debido a la razón de cardinalidad M:*N*.

En nuestro ejemplo, establecemos la transformación del tipo de vínculos M:*N* TRABAJA_EN de la figura 3.2 creando la relación TRABAJA_EN de la figura 6.5. La relación TRABAJA_EN recibe a veces el nombre de relación "vínculo", ya que corresponde a un tipo de vínculos. Incluimos las claves primarias de las relaciones PROYECTO y EMPLEADO como claves externas en TRABAJA_EN y cambiamos sus nombres a NÚMP y NSSE, respectivamente. También incluimos un atributo HORAS en TRABAJA_EN para representar el atributo Horas del tipo de vínculos. La clave primaria de la relación TRABAJA_EN es la combinación de los atributos de clave externa {NSSE, NÚMP}.

Cabe destacar que siempre es posible transformar los vínculos 1:1 o 1:*N* de una manera similar a como se hace con los vínculos M:*N*. Esta alternativa es útil sobre todo cuando hay pocos ejemplares del vínculo, a fin de evitar valores nulos en las claves externas. En este caso, la clave primaria de la relación "vínculo" será la clave externa de *sólo una* de las relaciones "entidad" participantes. En el caso de un vínculo 1:*N*, ésta será la relación entidad del lado *N*; en el caso de un vínculo 1:1, se elegirá la relación entidad con participación total (si existe).

PASO 6: Por cada atributo multivaluado *A* se crea una nueva relación *R* que contiene un atributo correspondiente a *A* más el atributo de clave primaria *K* (como clave externa en *R*) de la relación que representa el tipo de entidades o de vínculos que tiene a *A* como atributo. La clave primaria de *R* es la combinación de *A* y *K*. Si el atributo multivaluado es compuesto, se incluyen sus componentes simples.

En nuestro ejemplo, creamos una relación LUGARES_DEPTOS. El atributo LUGARD representa el atributo multivaluado Lugares de DEPARTAMENTO, en tanto que NÚMEROD —como clave externa— representa la clave primaria de la relación DEPARTAMENTO. La clave primaria de LUGARES_DEPTOS es la combinación de {NÚMEROD, LUGARD}. Habrá una tupla en LUGARES_DEPTOS por cada lugar en que esté ubicado un departamento.

La figura 6.5 muestra el esquema de base de datos relacional que se obtiene siguiendo los pasos anteriores, y la figura 6.6 presenta un ejemplar de muestra de la base de datos. Observe que no analizamos la transformación de los tipos de vínculos *n*-arios ($n > 2$) porque no hay ninguno en la figura 3.2; éstos pueden transformarse de manera similar a los tipos de vínculos M:*N* si se incluye el siguiente paso adicional en el procedimiento de transformación.

PASO 7: Por cada tipo de vínculos *n*-ario *R*, $n > 2$, se crea una nueva relación *S* que representa a *R*. Se incluyen como atributos de clave externa en *S* las claves primarias de las relaciones que representan los tipos de entidades participantes. También se incluyen los atributos simples (o los componentes simples de los atributos compuestos) del tipo de vínculos *n*-ario como atributos de *S*. La clave primaria de *S* casi siempre es una combinación de todas las claves externas que hacen referencia a las relaciones que representan los tipos de entidades participantes. No obstante, si la restricción de participación (*mín*, *máx*) de uno de los tipos de entidades *E* que participan en *R* tiene *máx* = 1, la clave primaria de *S* podrá ser el único atributo de clave externa que haga referencia a la relación *E'* que corresponde a *E*, la razón es que, en este caso, cada una de las entidades *e* de *E* participará en cuando más un ejemplar de vínculo de *R* y, por tanto, podrá identificar de manera única ese ejemplar. Con esto termina el procedimiento de transformación. ■

Por ejemplo, consideremos el tipo de vínculos SUMINISTRAR de la figura 3.16(a). Éste puede transformarse a la relación SUMINISTRAR que se muestra en la figura 6.19, cuya clave primaria es la combinación de claves externas {NOMPROV, NÚMCOMP, NOMPROY}.

Lo principal que debemos observar en un esquema relacional, en comparación con un esquema ER, es que los tipos de vínculos no se representan explícitamente; se representan mediante dos atributos *A* y *B*, uno como clave primaria y el otro como clave externa —con el mismo dominio— incluidos en dos relaciones *S* y *T*. Dos tuplas de *S* y *T* están relacionadas cuando tienen el mismo valor de *A* y *B*. Si usamos la operación EQUIRREUNIÓN (o REUNIÓN NATURAL) con *S.A* y *T.B*, podremos combinar todos los pares de tuplas relacionadas de *S* y *T* y materializar el vínculo. Cuando se trata de un tipo de vínculos binario 1:1 o 1:*N*, casi siempre se necesita una sola operación de reunión. En el caso de un tipo de vínculos binario M:*N*, se requieren dos operaciones de reunión, en tanto que para los tipos de vínculos *n*-arios se requieren *n* reuniones.

Por ejemplo, para formar una relación que incluya el nombre del empleado, el nombre del proyecto y el número de horas que el empleado trabaja en cada proyecto, necesitamos conectar cada tupla EMPLEADO con las tuplas PROYECTO relacionadas por la relación

PROVEEDOR

NOMPROV	. . .

PROYECTO

NOMPROY	. . .

COMPONENTE

NÚMCOMP	. . .

SUMINISTRAR

NOMPROV	NOMPROY	NÚMCOMP	CANTIDAD

Figura 6.19 Transformación del tipo de vínculos *n*-ario SUMINISTRAR de la figura 3.16(a).

TRABAJA_EN de la figura 6.5. Por tanto, deberemos aplicar la operación EQUIRREUNIÓN a las relaciones EMPLEADO y TRABAJA_EN con la condición de reunión NSS = NSSE, y luego aplicar otra EQUIRREUNIÓN a la relación resultante y la relación PROYECTO con la condición de reunión NÚMP = NÚMEROP. En general, cuando es preciso recorrer múltiples vínculos, hay que especificar muchas operaciones de reunión. El usuario de una base de datos relacional debe tener presentes siempre los atributos de clave externa para utilizarlos correctamente cuando combine tuplas relacionadas de dos o más relaciones.

La tabla 6.1 muestra los pares de atributos que se usan en operaciones de EQUIRREUNIÓN para materializar cada uno de los tipos de vínculos del esquema COMPAÑÍA que aparece en la figura 3.2. Para materializar el tipo de vínculos 1:*N* PERTENECE_A, aplicamos la EQUIRREUNIÓN a las relaciones EMPLEADO y DEPARTAMENTO con los atributos ND de EMPLEADO y NÚMEROD de DEPARTAMENTO. Según la base de datos de la figura 6.6, los empleados Silva, Vizcarra, Nieto y Esparza pertenecen al departamento 5 (Investigación); Zapata, Valdés y Jabbar pertenecen al departamento 4 (Administración), y Botello trabaja para el departamento 1 (Dirección).

Otro punto que debemos destacar en el esquema relacional es que creamos una relación individual por *cada* atributo multivaluado. Para una cierta entidad con un conjunto de valores para el atributo multivaluado, el valor del atributo clave de la entidad se repite una vez por cada valor del atributo multivaluado en una tupla individual. La razón es que el modelo relacional básico *no* permite valores (o conjuntos de valores) múltiples para un atributo en una sola tupla. Por ejemplo, como el departamento 5 está ubicado en tres lugares, hay tres tuplas en la relación LUGARES_DEPTOS de la figura 6.6; cada tupla especifica uno de los lugares. Se requiere una equirreunión para relacionar los valores del atributo multivaluado con los valores de otros atributos de una entidad o de un ejemplar de vínculo, pero aún así habrá múltiples tuplas. El álgebra relacional no cuenta con una operación ANIDAR o COMPRIMIR capaz de producir, a partir de la relación LUGARES_DEPTOS de la figura 6.6, un conjunto de tuplas de la forma {<1, Higueras>, <4, Santiago>, <5, {Belén, Sacramento, Higueras}>}. Ésta es una deficiencia grave de la actual versión normalizada o "plana" del

Tabla 6.1 Condiciones de reunión para materializar los tipos de vínculos del esquema ER COMPAÑÍA

Vínculo ER	Relaciones participantes	Condición de reunión
PERTENECE_A	EMPLEADO, DEPARTAMENTO	EMPLEADO.ND = DEPARTAMENTO.NÚMEROD
DIRIGE	EMPLEADO, DEPARTAMENTO	EMPLEADO.NSS = DEPARTAMENTO.NSSGTE
SUPERVISIÓN	EMPLEADO(E), EMPLEADO(S)	EMPLEADO(E).NSSSUPER = EMPLEADO(S).NSS
TRABAJA_EN	EMPLEADO, TRABAJA_EN PROYECTO	EMPLEADO.NSS = TRABAJA_EN .NSSE Y PROYECTO.NÚMEROP = TRABAJA_EN.NÚMP
CONTROLA	DEPARTAMENTO, PROYECTO	DEPARTAMENTO.NÚMEROD = PROYECTO.NÚMD
DEPENDIENTES_DE	EMPLEADO, DEPENDIENTE	EMPLEADO.NSS = DEPENDIENTE.NSSE

Tabla 6.2 Correspondencia entre los modelos ER y relacional

Modelo ER	Modelo relacional
tipo de entidades	relación "entidad"
tipo de vínculos 1:1 o 1:*N*	clave externa (o relación "vínculo")
tipo de vínculos M:*N*	relación "vínculo" y dos claves externas
tipo de vínculos *n*-ario	relación "vínculo" y *n* claves externas
atributo simple	atributo
atributo compuesto	conjunto de atributos componentes simples
atributo multivaluado	relación y clave externa
conjunto de valores	dominio
atributo clave	clave primaria (o secundaria)

modelo relacional. Los lenguajes relacionales SQL, QUEL y QBE tampoco cuentan con mecanismos para manejar esta clase de conjuntos de valores dentro de las tuplas. En este aspecto, los modelos orientados a objetos, jerárquico y de red proporcionan mejores recursos que el modelo relacional. El modelo relacional anidado (véase el Cap. 21) intenta remediar esto.

En nuestro ejemplo, aplicamos una EQUIRREUNIÓN a LUGARES_DEPTOS y DEPARTAMENTO con base en el atributo NÚMEROD para obtener los valores de todos los lugares junto con otros atributos de DEPARTAMENTO. En la relación resultante, los valores de los otros atributos de departamento se repiten en tuplas individuales por cada lugar de cada departamento.

6.8.2 *Resumen de la transformación de elementos y restricciones del modelo*

En la tabla 6.2 se resumen las correspondencias entre los elementos y restricciones del modelo ER y el modelo relacional.

6.9 Resumen

En este capítulo presentamos los conceptos de modelado que ofrece el modelo relacional de los datos. También examinamos el álgebra relacional y las operaciones adicionales con que podemos manipular relaciones. Comenzamos por explicar los conceptos de dominio, tupla y atributo, para luego definir un esquema de relación como una lista de atributos que describen la estructura de una relación. Una relación, o un ejemplar de relación, es un conjunto de tuplas.

Las relaciones se distinguen de las tablas o archivos ordinarios por varias características. La primera es que las tuplas de una relación no están ordenadas. La segunda consiste en el ordenamiento de los atributos en un esquema de relación y el ordenamiento correspondiente de los valores dentro de una tupla. Dimos una definición alternativa de relación que no requiere estos dos ordenamientos, pero, por comodidad, seguimos usando la primera definición, que exige que los atributos y valores de las tuplas estén ordenados. Después analizamos los valores de las tuplas, así como los valores nulos que representan información faltante o desconocida.

También definimos los esquemas de bases de datos relacionales y las bases de datos relacionales. Varios tipos de restricciones pueden considerarse como parte del modelo relacional. Los conceptos de superclave, clave candidata y clave primaria especifican las restricciones de clave. La restricción de integridad de entidades prohíbe los valores nulos en los atributos

Tabla 6.3 Operaciones del álgebra relacional

Operación	Propósito	Notación
SELECCIONAR	Selecciona todas las tuplas de una relación R que satisfagan la condición de selección.	$\sigma_{<\text{condición de selección}>}(R)$
PROYECTAR	Produce una nueva relación con sólo algunos de los atributos de R, y elimina tuplas repetidas.	$\pi_{<\text{lista de atributos}>}(R)$
REUNIÓN THETA	Produce todas las combinaciones de tuplas de R_1 y R_2 que satisfagan la condición de reunión.	$R_1 \bowtie_{<\text{condición de reunión}>} R_2$
EQUIRREUNIÓN	Produce todas las combinaciones de tuplas de R_1 y R_2 que satisfagan una condición de reunión que sólo tiene comparaciones de igualdad.	$R_1 \bowtie_{<\text{condición de reunión}>} R_2$, o $R_1 \bowtie_{(<\text{atributos de reunión 1}>),\ (<\text{atributos de reunión 2}>)} R_2$
REUNIÓN NATURAL	Igual que la EQUIRREUNIÓN excepto que los atributos de reunión de R_2 no se incluyen en la relación resultante; si los atributos de reunión tienen los mismos nombres, no es preciso especificarlos.	$R_1 *_{<\text{condición de reunión}>} R_2$, o $R_1 *_{(<\text{atributos de reunión 1}>),\ (<\text{atributos de reunión 2}>)} R_2$, o $R_1 * R_2$
UNIÓN	Produce una relación que incluye todas las tuplas que están en R_1 o en R_2, o en ambas; R_1 y R_2 deben ser compatibles con la unión.	$R_1 \cup R_2$
INTERSECCIÓN	Produce una relación que incluye todas las tuplas que están tanto en R_1 como en R_2; R_1 y R_2 deben ser compatibles con la unión.	$R_1 \cap R_2$
DIFERENCIA	Produce una relación que incluye todas las tuplas que están en R_1 y que no están en R_2; R_1 y R_2 deben ser compatibles con la unión.	$R_1 - R_2$
PRODUCTO CARTESIANO	Produce una relación que tiene los atributos de R_1 y R_2 e incluye como tuplas todas las posibles combinaciones de tuplas de R_1 y R_2.	$R_1 \times R_2$
DIVISIÓN	Produce una relación $R(X)$ que incluye todas las tuplas $t[X]$ de $R_1(Z)$ que aparecen en R_1 en combinación con todas las tuplas de $R_2(Y)$, donde $Z = X \cup Y$.	$R_1(Z) \div R_2(Y)$

de clave primaria. La restricción de integridad referencial entre relaciones sirve para mantener la consistencia de las referencias entre tuplas de diferentes relaciones.

Las operaciones de actualización del modelo relacional son insertar, eliminar y modificar; todas ellas pueden violar ciertos tipos de restricciones. Siempre que se aplique una actualización, es preciso comprobar la consistencia de la base de datos después de la operación para asegurarse de no haber violado alguna restricción. Los esquemas relacionales se declaran mediante un DDL relacional.

El álgebra relacional es un conjunto de operaciones para manipular relaciones. Presentamos las diversas operaciones e ilustramos los tipos de manipulaciones que podemos efectuar con cada una. La tabla 6.3 es una lista de las diversas operaciones del álgebra relacional que vimos. Primero estudiamos los operadores relacionales unarios SELECCIONAR y PROYECTAR, y luego las operaciones binarias de conjuntos que exigen que las relaciones que las que se aplican

sean compatibles con la unión; entre ellas están UNIÓN, INTERSECCIÓN y DIFERENCIA. La operación PRODUCTO CARTESIANO sirve para combinar tuplas de dos relaciones para formar tuplas más grandes. Vimos cómo un PRODUCTO CARTESIANO seguido de SELECCIONAR puede identificar las tuplas relacionadas de dos relaciones. Las operaciones de REUNIÓN pueden identificar y combinar directamente las tuplas relacionadas; entre ellas están la REUNIÓN THETA, la EQUIRREUNIÓN y la REUNIÓN NATURAL.

A continuación analizamos algunos tipos de consultas que no se pueden expresar con las operaciones del álgebra relacional. Presentamos la operación FUNCIÓN para manejar las solicitudes de tipo agregado; analizamos las consultas recursivas y vimos cómo especificar algunos tipos de éstas; luego presentamos las operaciones REUNIÓN EXTERNA y UNIÓN EXTERNA que extienden la REUNIÓN y la UNIÓN.

En la sección 6.7 dimos ejemplos adicionales que ilustran el empleo de las operaciones relacionales para especificar consultas en una base de datos relacional. Usaremos estas consultas en capítulos subsecuentes cuando analicemos diversos lenguajes de consulta.

En la sección 6.8 mostramos cómo transformar un diseño de esquema conceptual en el modelo ER a un esquema de base de datos relacional. Dimos un algoritmo para la transformación ER-modelo relacional y lo ilustramos con ejemplos de la base de datos COMPAÑÍA. La tabla 6.2 resume las correspondencias entre los elementos y restricciones de los modelos ER y relacional.

Preguntas de repaso

6.1. Defina los siguientes términos: *dominio, atributo, n-tupla, esquema de relación, ejemplar de relación, grado de una relación, esquema de base de datos relacional, ejemplar de base de datos relacional.*

6.2. ¿Por qué no están ordenadas las tuplas de una relación?

6.3. ¿Por qué no se permiten tuplas repetidas en una relación?

6.4. ¿Qué diferencia hay entre una clave y una superclave?

6.5. ¿Por qué designamos una de las claves candidatas de una relación como clave primaria?

6.6. Analice las características de las relaciones que las distinguen de las tablas y archivos ordinarios.

6.7. Explique por qué pueden aparecer valores nulos en las relaciones.

6.8. Analice las restricciones de integridad de entidades y de integridad referencial. ¿Por qué se consideran importantes?

6.9. Defina las *claves externas*. ¿Para qué nos sirve este concepto?

6.10. Analice las diversas operaciones de actualización de relaciones y los tipos de restricciones de integridad que debemos verificar en cada operación de actualización.

6.11. Mencione las operaciones del álgebra relacional y el propósito de cada una de ellas.

6.12. ¿Qué significa compatibilidad con la unión? ¿Por qué las operaciones UNIÓN, INTERSECCIÓN y DIFERENCIA requieren que las relaciones a las que se aplican sean compatibles con la unión?

6.13. Analice algunos tipos de consultas en las que sea necesario cambiar los nombres de los atributos para especificar la consulta sin ambigüedad.

6.14. Analice los diversos tipos de operaciones de REUNIÓN.

6.15. ¿Qué es la operación FUNCIÓN? ¿Para qué sirve?

6.16. ¿En qué se distinguen las operaciones de REUNIÓN EXTERNA y las de REUNIÓN? ¿Qué diferencia hay entre la operación de UNIÓN EXTERNA y la de UNIÓN?

6.17. Explique las correspondencias entre los elementos del modelo ER y los del modelo relacional. Indique la forma de transformar cada uno de los elementos del modelo ER al modelo relacional, y analice cualesquier transformaciones alternativas que existan.

Ejercicios

6.18. Muestre el resultado de aplicar las consultas de ejemplo de la sección 6.7 a la base de datos de la figura 6.6.

6.19. Especifique las siguientes consultas en términos del esquema de base de datos de la figura 6.5, empleando los operadores relacionales que vimos en este capítulo. Muestre además el resultado de aplicar cada consulta a la base de datos de la figura 6.6.

a. Obtenga los nombres de todos los empleados del departamento 5 que trabajan más de 10 horas por semana en el proyecto 'ProductoX'.

b. Cite los nombres de todos los empleados que tienen un dependiente con el mismo nombre de pila que ellos.

c. Encuentre los nombres de todos los empleados supervisados directamente por 'Federico Vizcarra'.

d. Para cada proyecto, cite el nombre del proyecto y el total de horas que trabajan todos los empleados en ese proyecto.

e. Obtenga los nombres de todos los empleados que trabajan en cada uno de los proyectos.

f. Obtenga los nombres de los empleados que no trabajan en ningún proyecto.

g. Para cada departamento, obtenga el nombre del departamento y el salario medio de todos los empleados que trabajan en él.

h. Obtenga el salario medio de todos los empleados de sexo femenino.

i. Encuentre los nombres y direcciones de todos los empleados que trabajan en, por lo menos, un proyecto situado en Higueras pero cuyo departamento no está ubicado ahí.

j. Prepare una lista con los apellidos de todos los gerentes de departamento que no tienen dependientes.

6.20. Suponga que se aplican las siguientes operaciones de actualización directamente a la base de datos de la figura 6.6. Analice *todas* las restricciones de integridad que viola cada operación, si lo hace, y las diferentes formas de imponer dichas restricciones.

a. Insertar <'Roberto', 'F', 'Saldaña', '943775543', '21-JUN-42', '2365 Ave. Naranjos, Belén, MX', M, 58000, '888665555', 1> en EMPLEADO.

b. Insertar <'ProductoA', 4, 'Belén', 2> en PROYECTO.

c. Insertar <'Producción', 4, '943775543', '01-OCT-88'> en DEPARTAMENTO.

d. Insertar <'677678989', nulo, '40.0'> en TRABAJA_EN.

e. Insertar <'453453453', 'José', M, '12-DIC-60','CÓNYUGE'> en DEPENDIENTE.

f. Eliminar las tuplas de TRABAJA_EN con NSSE = '333445555'.

g. Eliminar la tupla EMPLEADO con NSS = '987654321'.

h. Eliminar la tupla PROYECTO con NOMBREP = 'ProductoX'.

i. Modificar NSSGTE y FECHAINICGTE de la tupla DEPARTAMENTO con NÚMEROD = 5 cambiándolos a '123456789' y '01-OCT-93', respectivamente.

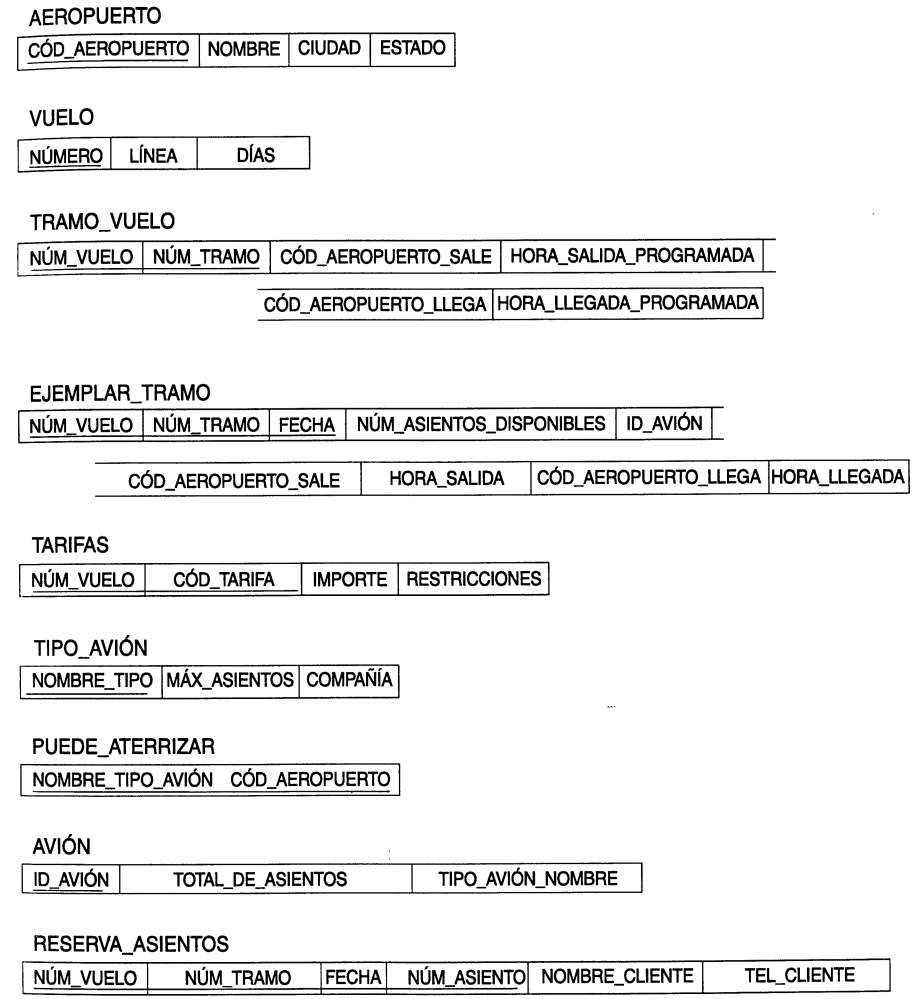


Figura 6.20 El esquema de base de datos relacional AEROLÍNEA.

j. Modificar el atributo NSSSUPER de la tupla EMPLEADO con NSS = '999887777' cambiándolo a '943775543'.

k. Modificar el atributo HORAS de la tupla TRABAJA_EN con NSSE = '999887777' y NÚMP = 10 cambiándolo a '5.0'.

6.21. Considere el esquema de base de datos relacional AEROLÍNEA que se muestra en la figura 6.20, y que describe una base de datos con información sobre vuelos de líneas aéreas. Cada VUELO se identifica con un NÚMERO de vuelo, y consta de uno o más TRAMO_VUELO con NÚM_TRAMO 1, 2, 3, etc. Cada tramo tiene horas y aeropuertos de salida y llegada programados, y tiene muchos EJEMPLAR_TRAMO, uno por cada FECHA en que tiene lugar el vuelo. Se mantienen TARIFAS para cada vuelo. Para cada ejemplar de tramo, se mantienen RESERVA_ASIENTOS, el AVIÓN empleado en el tramo y las horas de salida y llegada y los aeropuertos específicos. Un AVIÓN se identifica con ID_AVIÓN y es de un cierto TIPO_AVIÓN. PUEDE_ATERRIZAR relaciona los TIPO_AVIÓN con los AEROPUERTO en los que pueden aterrizar. Cada AEROPUERTO se identifica con un CÓD_AEROPUERTO. Especifique las siguientes consultas en álgebra relacional:

a. Para cada vuelo, prepare una lista con el número de vuelo, el aeropuerto de salida del primer tramo del vuelo y el aeropuerto de llegada del último tramo del vuelo.

b. Prepare una lista con los números de vuelo y los días de todos los vuelos o tramos de vuelo que salen del aeropuerto Houston Intercontinental (código de aeropuerto 'IAH') y llegan al aeropuerto internacional de Los Angeles (código 'LAX').

c. Prepare una lista con los números de vuelo, códigos de aeropuerto de salida, horas de salida programadas, códigos de aeropuerto de llegada, horas de llegada programadas y días de todos los vuelos o tramos de vuelo que salgan de algún aeropuerto de la ciudad de Houston y lleguen a algún aeropuerto de la ciudad de Los Angeles.

d. Obtenga toda la información de tarifas del vuelo número 'CO197'.

e. Obtenga el número de asientos disponibles en el vuelo número 'CO197' del '09-OCT-95'.

6.22. Considere una actualización de la base de datos AEROLÍNEA para introducir una reservación en un cierto vuelo o tramo de vuelo en una fecha dada.

a. Indique las operaciones para esta actualización.

b. ¿Qué tipos de restricciones verificaría?

c. ¿Cuáles de esas restricciones son de clave, de integridad de entidades y de integridad referencial, y cuáles no?

d. Especifique todas las restricciones de integridad referencial de la figura 6.20.

6.23. Considere la relación

CLASE(NúmCurso, NúmSeccUniv, NombreProf, Semestre, CódEdificio, NúmSalón, Hora, Días, HorasCrédito).

Ésta representa clases impartidas en una universidad, con NúmSeccUniv único. Identifique las que podrían ser claves candidatas y describa con sus propias palabras las restricciones para que sea válida cada una de esas claves candidatas.

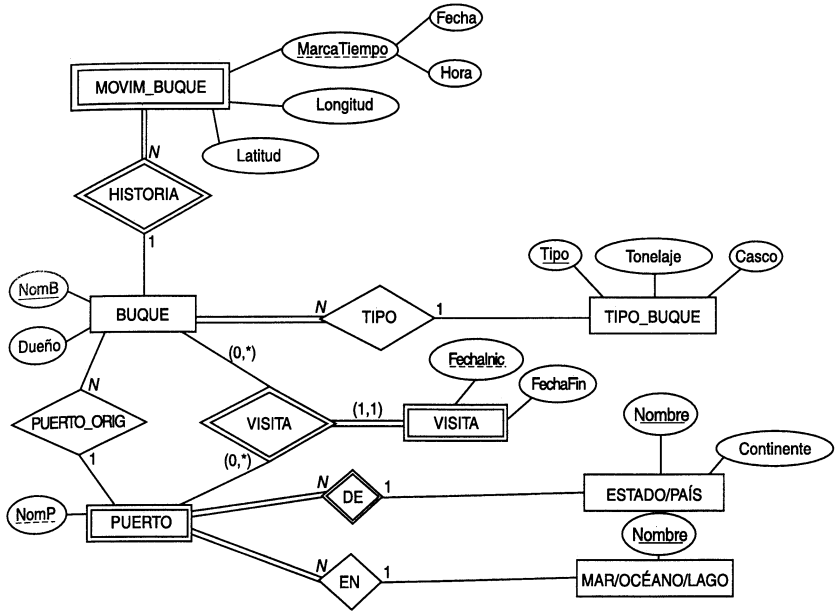


Figura 6.21 Esquema ER para una base de datos CONTROL_BUQUES.

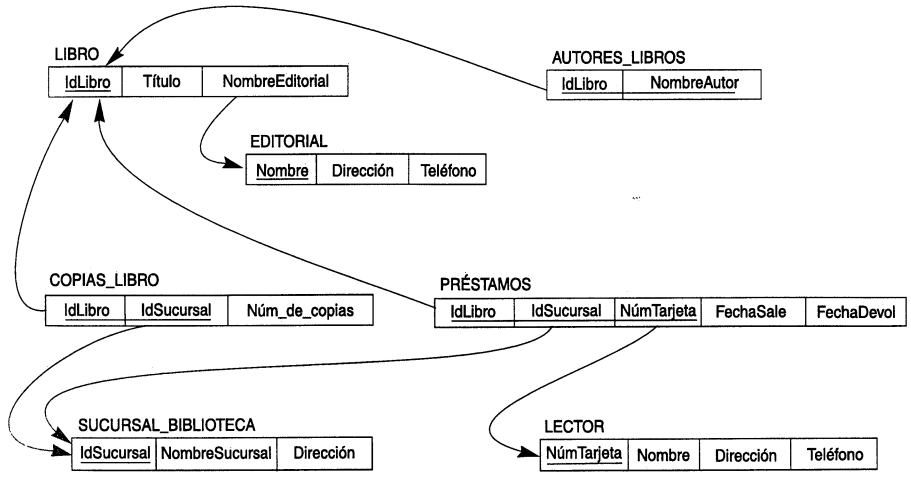


Figura 6.22 Esquema de base de datos relacional para una base de datos BIBLIOTECA.

6.24. La figura 6.21 muestra un esquema ER para una base de datos que serviría para llevar el control de buques de carga y su ubicación para las autoridades marítimas. Transforme este esquema a un esquema relacional y especifique todas las claves primarias y externas.

6.25. Transforme el esquema ER BANCO del ejercicio 3.23 (Fig. 3.20) a un esquema relacional. Especifique todas las claves primarias y externas.

6.26. Considere el esquema relacional BIBLIOTECA que se muestra en la figura 6.22 y que sirve para llevar el control de libros, lectores y préstamos de libros. Las restricciones de integridad referencial se indican con arcos dirigidos, según la notación de la figura 6.7. Escriba expresiones relacionales para las siguientes consultas de la base de datos BIBLIOTECA:

a. ¿Cuántas copias del libro intitulado *La tribu perdida* posee la sucursal "Salvatierra" de la biblioteca?

b. ¿Cuántas copias del libro *La tribu perdida* posee cada una de las sucursales de la biblioteca?

c. Obtenga los nombres de todos los lectores que no tengan libros en préstamo.

d. Para cada libro prestado por la sucursal "Salvatierra" cuya fecha de devolución (FechaDevol) sea la de hoy, obtenga el título del libro, el nombre del lector y la dirección del lector.

e. Para cada sucursal de la biblioteca, obtenga su nombre y el número total de libros que tiene en préstamo.

f. Para todos los lectores que tienen más de cinco libros en préstamo, obtenga sus nombres, sus direcciones y el número de libros.

g. Para cada libro escrito (total o parcialmente) por "Stephen King", obtenga el título y el número de copias que posee la sucursal de nombre "Central".

6.27. Intente transformar el esquema relacional de la figura 6.22 a un esquema ER. Esto es parte de un proceso llamado *ingeniería inversa*, mediante el cual se crea un esquema conceptual a partir de una base de datos ya implementada. Exprese todas las suposiciones que haga.

Bibliografía selecta

Codd (1970) presentó el modelo relacional en un artículo clásico; también introdujo el álgebra relacional y estableció las bases teóricas del modelo relacional en una serie de artículos (Codd 1971, 1972, 1972*a*, 1974). Después fue galardonado con el premio Turing, el reconocimiento máximo de la ACM, por sus trabajos sobre el modelo relacional. En un artículo posterior, Codd (1979) propuso extender el modelo relacional e incorporar valores NULO en el álgebra relacional. El modelo resultante se conoce como RM/T. Trabajos anteriores realizados por Childs (1968) habían utilizado la teoría de conjuntos para modelar bases de datos.

Se han efectuado muchas investigaciones sobre diversos aspectos del modelo relacional. Todd (1976) describe un SGBD experimental que implementa directamente las operaciones del álgebra relacional. Date (1983*a*) analiza las reuniones externas. Schmidt y Swenson (1975) propusieron una semántica adicional para el modelo relacional al clasificar diferentes tipos de relaciones. Wiederhold y Elmasri (1979) introdujeron diversos tipos de conexiones entre las relaciones para mejorar sus restricciones. Los trabajos encaminados a extender el modelo relacional se analizan en Carlis (1986) y en Ozsoyoglu *et al.* (1985). Cammarata *et al.* (1989) extienden las restricciones de integridad y las reuniones del modelo relacional. En los capítulos 7, 8, 9, 12, 13, 16, 20, 21 y 23 se proporcionan notas bibliográficas adicionales sobre otros aspectos del modelo relacional y sus lenguajes, sistemas, extensiones y teoría.

CAPÍTULO 7

SQL: un lenguaje de bases de datos relacionales

En el capítulo 6 estudiamos las operaciones del álgebra relacional, imprescindibles para entender los tipos de solicitudes que podemos especificar en una base de datos relacional. También son importantes para procesar y optimizar las consultas en un SGBD relacional, como veremos en el capítulo 16. En general, el álgebra relacional se clasifica como un lenguaje de consulta de alto nivel porque sus operaciones se aplican a relaciones completas. No obstante, son muy pocos los lenguajes comerciales de SGBD que se basan directamente en el álgebra relacional.[†] La razón es que las consultas del álgebra relacional se escriben en forma de secuencias de operaciones que, al ejecutarse, producen el resultado deseado. Al especificar una consulta en el álgebra relacional, el usuario debe indicar *cómo* —en qué orden— se deben ejecutar las operaciones de consulta. La mayor parte de los SGBD relacionales que se encuentran en el mercado cuentan con una interfaz de lenguaje *declarativo* de alto nivel, de modo que el usuario sólo tenga que especificar *cuál* es el resultado deseado, dejando que el SGBD se encargue de la optimización efectiva y de las decisiones sobre cómo se ejecutará la consulta.

En este capítulo y en el 8 analizaremos varios lenguajes, implementados parcial o totalmente, que están disponibles en SGBD comerciales. El mejor conocido de ellos es SQL, cuyo nombre se deriva de *Structured Query Language* (lenguaje estructurado de consulta). En el capítulo 8 presentaremos otros dos lenguajes, QUEL y QBE, después de estudiar el cálculo relacional, que es el lenguaje formal en el que se basan QUEL y QBE (y, en cierta medida, SQL).

Originalmente, SQL se llamaba SEQUEL (por *Structured English QUEry Language*: lenguaje estructurado de consultas en inglés) y se diseñó e implementó en IBM Research como

[†]Uno de los primeros SGBD experimentales, llamado ISBL, implementaba las operaciones del álgebra relacional como lenguaje de consulta propio (Todd 1976).

interfaz para un sistema experimental de bases de datos relacionales llamado SYSTEM R. Ahora SQL es el lenguaje de los SGBD relacionales comerciales DB2 y SQL/DS de IBM, y fue el primero de los lenguajes de bases de datos de alto nivel junto con QUEL. Casi todos los proveedores de SGBD comerciales han implementado variaciones de SQL, y un esfuerzo conjunto que realizan actualmente ANSI (American National Standards Institute) e ISO (International Standards Organization) ha dado lugar a una versión estándar de SQL (ANSI 1986), llamada SQL1. También se ha creado ya una norma revisada y muy expandida, llamada SQL2 (o bien SQL–92), y ya existen planes para un SQL3 que ampliará aún más el lenguaje con conceptos de orientación a objetos y otros conceptos recientes de bases de datos.

SQL es un lenguaje de base de datos completo; cuenta con enunciados de definición, consulta y actualización de datos. Así pues, es *tanto* un DDL *como* un DML. Por añadidura, cuenta con mecanismos para definir vistas de la base de datos, crear y desechar índices de los archivos que representan relaciones (aunque en SQL2 éstos ya no existen) y para incorporar enunciados de SQL en lenguajes de programación de propósito general como C o PASCAL. Trataremos estos temas en las subsecciones que siguen, apegándonos en general a SQL2. No obstante, describiremos algunas características, como CREATE INDEX, que se excluyeron de SQL2 y que se basan en versiones anteriores de SQL, pues muchos SGBD relacionales todavía cuentan con ellas. En el capítulo 15 haremos un breve análisis de los recursos de catálogo y diccionario de SQL, y en el capítulo 20 ofreceremos un panorama de las órdenes para la autorización de privilegios en SQL.

Si el lector desea una introducción menos exhaustiva a SQL, puede pasar por alto una parte o la totalidad de las siguientes secciones: 7.2.5 a 7.2.9, 7.5, 7.6 y 7.7.

7.1 Definición de datos en SQL

SQL emplea los términos **tabla** (*table*), **fila** (*row*) y **columna** (*column*) en vez de relación, tupla y atributo, respectivamente. Nosotros usaremos de manera indistinta los términos correspondientes. Las órdenes de SQL para definir los datos son CREATE (crear), ALTER (alterar) y DROP (desechar); éstas las veremos en las secciones 7.1.2 a 7.1.4. Sin embargo, antes de hacerlo analizaremos los conceptos de esquema y catálogo. Sólo presentaremos una sinopsis de las características más importantes; los detalles pueden consultarse en el documento de SQL2.

7.1.1 *Conceptos de esquema y catálogo en SQL2*

Las primeras versiones de SQL no contemplaban el concepto de esquema de base de datos relacional; todas las tablas (relaciones) se consideraban parte del mismo esquema. El concepto de esquema SQL se incorporó en SQL2 con el fin de agrupar tablas y otros elementos pertenecientes a la misma aplicación de bases de datos. Un **esquema SQL** se identifica con un **nombre de esquema**, y consta de un identificador de autorización que indica el usuario o la cuenta que es propietario(a) del esquema, además de los **descriptores** de *cada elemento* del esquema. Dichos elementos comprenden tablas, vistas, dominios y otros (como las concesiones de autorización y las aserciones), que describen el esquema. Los esquemas se crean mediante la instrucción CREATE SCHEMA, que puede contener todas las definiciones de los elementos del esquema. Como alternativa, podemos asignar un nombre y un identificador de autorización al esquema, y definir los elementos más adelante. Por ejemplo, la siguiente

instrucción crea un esquema llamado COMPAÑÍA, cuyo propietario es el usuario con identificador de autorización JSILVA:

CREATE SCHEMA COMPAÑÍA **AUTHORIZATION** JSILVA;

Además del concepto de esquema, SQL2 emplea el concepto de **catálogo**: una colección nombrada de esquemas en un entorno SQL. Todo catálogo contiene un esquema especial llamado INFORMATION_SCHEMA, que proporciona a los usuarios autorizados información sobre todos los descriptores de elementos de todos los esquemas en el catálogo. Se pueden definir restricciones de integridad entre relaciones, como la integridad referencial, sólo si dichas relaciones existen en esquemas dentro del mismo catálogo. Además, los esquemas del mismo catálogo pueden compartir ciertos elementos, como las definiciones de dominios.

7.1.2 *La orden CREATE TABLE y los tipos de datos y restricciones de SQL*

La orden CREATE TABLE sirve para especificar una nueva relación dándole un nombre y especificando sus atributos y restricciones. Los atributos se especifican primero, y a cada uno se le da un nombre, un tipo de datos para especificar su dominio de valores, y quizá algunas restricciones. En seguida se especifican las restricciones de clave, de integridad de entidades y de integridad referencial. La figura 7.1(a) muestra ejemplos de enunciados de definición de datos en SQL para el esquema de base de datos relacional de la figura 6.7. Por lo regular, el esquema SQL en el que se declaran las relaciones se especifica implícitamente por el entorno en que se ejecutan los enunciados CREATE TABLE. Como alternativa, podemos anexar explícitamente el nombre del esquema al nombre de la relación, separándolos con un punto. Por ejemplo, si escribimos la orden

CREATE TABLE COMPAÑÍA.EMPLEADO...

en vez de

CREATE TABLE EMPLEADO...

como en la figura 7.1(a), podemos declarar explícitamente que la tabla EMPLEADO forma parte del esquema SQL COMPAÑÍA.

Entre los **tipos de datos** disponibles para los atributos están los numéricos, los de cadena de caracteres, los de cadena de bits y los de fecha y hora. Los tipos de datos **numéricos** incluyen números enteros de diversos tamaños (INTEGER o INT, y SMALLINT) y números reales de diversas precisiones (FLOAT, REAL, DOUBLE PRECISION). Podemos declarar números con formato empleando DECIMAL(*i,j*) (o DEC(*i,j*) o NUMERIC(*i,j*)), donde *i*, la *precisión*, es el número total de dígitos decimales y *j*, la *escala*, es el número de dígitos que aparecen después del punto decimal. El valor por omisión de la escala es cero, y el valor por omisión de la precisión depende de la implementación.

Los tipos de datos de **cadena de caracteres** tienen longitud fija (CHAR(*n*) o CHARACTER (*n*), donde *n* es el número de caracteres) o variable (VARCHAR(*n*) o CHAR VARYING(*n*) o CHARACTER VARYING(*n*), donde *n* es el número máximo de caracteres). Los tipos de datos de **cadena de bits** tienen longitud fija *n* (BIT(*n*)) o variable (BIT VARYING(*n*), donde *n* es el número máximo de bits). El valor por omisión de *n*, la longitud de una cadena de caracteres o de bits, es uno.

SQL2 cuenta con nuevos tipos de datos para **fecha** y **hora**. El tipo de datos DATE (fecha) tiene diez posiciones, y sus componentes son YEAR, MONTH y DAY (año, mes y día,

(a)

```
CREATE TABLE EMPLEADO
        ( NOMBREP           VARCHAR(15)     NOT NULL,
          INIC              CHAR,
          APELLIDO          VARCHAR(15)     NOT NULL,
          NSS               CHAR(9)         NOT NULL,
          FECHAN            DATE,
          DIRECCIÓN         VARCHAR(30),
          SEXO              CHAR,
          SALARIO           DECIMAL(10,2),
          NSSSUPER          CHAR(9),
          ND                INT             NOT NULL,
    PRIMARY KEY (NSS),
    FOREIGN KEY (NSSSUPER) REFERENCES EMPLEADO(NSS),
    FOREIGN KEY (ND) REFERENCES DEPARTAMENTO(NÚMEROD));
CREATE TABLE DEPARTAMENTO
        ( NOMBRED           VARCHAR(15)     NOT NULL,
          NÚMEROD           INT             NOT NULL,
          NSSGTE            CHAR(9)         NOT NULL,
          FECHAINICGTE      DATE,
        PRIMARY KEY (NÚMEROD),
        UNIQUE (NOMBRED),
        FOREIGN KEY (NSSGTE) REFERENCES EMPLEADO(NSS));
CREATE TABLE LUGARES_DEPTOS
        ( NÚMEROD           INT             NOT NULL,
          LUGARD            VARCHAR(15)     NOT NULL,
        PRIMARY KEY (NÚMEROD, LUGARD),
        FOREIGN KEY (NÚMEROD) REFERENCES DEPARTAMENTO(NÚMEROD));
CREATE TABLE PROYECTO
        ( NOMBREPR          VARCHAR(15)     NOT NULL,
          NÚMEROP           INT             NOT NULL,
          LUGARP            VARCHAR(15),
          NÚMD              INT             NOT NULL,
        PRIMARY KEY (NÚMEROP),
        UNIQUE (NOMBREPR),
        FOREIGN KEY (NÚMD) REFERENCES DEPARTAMENTO(NÚMEROD));
CREATE TABLE TRABAJA_EN
        ( NSSE              CHAR(9)         NOT NULL,
          NÚMP              INT             NOT NULL,
          HORAS             DECIMAL(3,1)    NOT NULL,
        PRIMARY KEY (NSSE, NÚMP),
        FOREIGN KEY (NSSE) REFERENCES EMPLEADO(NSS),
        FOREIGN KEY (NÚMP) REFERENCES PROYECTO(NÚMEROP));
CREATE TABLE DEPENDIENTE
        ( NSSE      CHAR(9)                 NOT NULL,
          NOMBRE_DEPENDIENTE  VARCHAR(15)   NOT NULL,
          SEXO      CHAR,
          FECHAN    DATE,
          PARENTESCO  VARCHAR(8),
        PRIMARY KEY (NSSE, NOMBRE_DEPENDIENTE),
        FOREIGN KEY (NSSE) REFERENCES EMPLEADO(NSS));
```

Figura 7.1 Definiciones de datos en SQL2. (a) Instrucciones de SQL2 que definen el esquema COMPAÑÍA de la figura 6.7. (b) Especificación de acciones disparadas por integridad referencial.

```
(b)
CREATE TABLE EMPLEADO
        (. . . ,
          ND        INT   NOT NULL              DEFAULT 1,
        CONSTRAINT CLPEMP
          PRIMARY KEY (NSS),
        CONSTRAINT CLESUPEREMP
          FOREIGN KEY (NSSSUPER) REFERENCES EMPLEADO(NSS)
                        ON DELETE SET NULL      ON UPDATE CASCADE,
        CONSTRAINT CLEDEPTOEMP
          FOREIGN KEY (ND) REFERENCES DEPARTAMENTO(NÚMEROD)
                        ON DELETE SET DEFAULT  ON UPDATE CASCADE);

CREATE TABLE DEPARTAMENTO
        (. . . ,
          NSSGTE   CHAR(9)  NOT NULL  DEFAULT "888665555",
        . . . ,
         CONSTRAINT CLPDEPTO
          PRIMARY KEY (NÚMEROD),
        CONSTRAINT CLSDEPTO
          UNIQUE (NOMBRED),
        CONSTRAINT CLEGTEDEPTO
          FOREIGN KEY (NSSGTE) REFERENCES EMPLEADO(NSS)
                ON DELETE SET DEFAULT   ON UPDATE CASCADE);

CREATE TABLE LUGARES_DEPTOS
        (. . . ,
        PRIMARY KEY (NÚMEROD, LUGARD),
        FOREIGN KEY (NÚMEROD) REFERENCES DEPARTAMENTO(NÚMEROD)
          ON DELETE CASCADE ON UPDATE CASCADE);
```

Figura 7.1 *(continuación)*

respectivamente), por lo regular en la forma YYYY-MM-DD. El tipo de datos TIME (hora) tiene por lo menos ocho posiciones, con los componentes HOUR, MINUTE y SECOND (hora, minuto y segundo), normalmente en la forma HH-MM-SS. La implementación de SQL deberá permitir sólo fechas y horas válidas. Además, un tipo de datos TIME(*i*), donde *i* es la *precisión de fracciones de segundo*, especifica *i* + 1 posiciones adicionales para TIME: una posición para un carácter separador adicional e *i* posiciones para especificar fracciones decimales de segundo. Un TIME WITH TIME ZONE (hora con huso horario) contiene seis posiciones extra para especificar el *desplazamiento* respecto al huso horario estándar universal, que está en el intervalo de +13:00 a –12:59 en unidades de HOURS:MINUTES. Si no se incluye WITH TIME ZONE, el valor por omisión es el huso horario local de la sesión SQL. Por último, un tipo de datos de **marca de tiempo** (TIMESTAMP) incluye los campos DATE y TIME, más un mínimo de seis posiciones para fracciones de segundo y un calificador opcional WITH TIME ZONE.

Otro tipo de datos relacionado con DATE, TIME y TIMESTAMP es INTERVAL, el cual especifica un **intervalo**: un *valor relativo* que puede servir para incrementar o decrementar un valor absoluto de fecha, hora o marca de tiempo. Los intervalos se califican con YEAR/MONTH (año/mes) o con DAY/TIME (día/hora) para indicar su naturaleza.

En SQL2 es posible especificar directamente el tipo de datos de cada atributo, como en la figura 7.1(a); una alternativa es declarar un dominio y usar su nombre. Esto hace más fácil cambiar el tipo de datos de un dominio utilizado por un gran número de atributos de un esquema, y hace que este último sea más comprensible. Por ejemplo, podemos crear un dominio TIPO_NSS con la siguiente instrucción:

```
CREATE DOMAIN TIPO_NSS AS CHAR(9);
```

Podemos usar TIPO_NSS en vez de CHAR(9) en la figura 7.1(a) para los atributos NSS y NSSSUPER de EMPLEADO, NSSGTE de DEPARTAMENTO, NSSE de TRABAJA_EN y NSSE de DEPENDIENTE. Los dominios también pueden tener especificaciones por omisión opcionales mediante una cláusula DEFAULT, como veremos cuando hablemos de los atributos.

Dado que SQL permite NULL (nulo) como valor de un atributo, se puede especificar una *restricción* NOT NULL si no se permiten valores nulos para ese atributo. Se debe especificar siempre esta restricción para los atributos de clave primaria de toda relación, así como para cualquier otro atributo cuyos valores no deban ser nulos, como se aprecia en la figura 7.1(a). También es posible definir un *valor por omisión* para un atributo anexando la cláusula DEFAULT <valor> a la definición de un atributo. El valor por omisión se incluirá en todas las tuplas nuevas si no se proporciona un valor explícito para ese atributo. La figura 7.1(b) ilustra la especificación de un gerente por omisión para un nuevo departamento y un departamento por omisión para un nuevo empleado. Si no se especifica la cláusula DEFAULT, el valor por omisión por omisión (¡!) será NULO.

Después de las especificaciones de atributos (o columnas), podemos especificar *restricciones de tabla* adicionales, incluidas las de clave y de integridad referencial, como se ilustra en la figura 7.1(a).† La cláusula PRIMARY KEY especifica uno o más atributos que constituyen la clave primaria de una relación. La cláusula UNIQUE (único) especifica otras posibles claves. La integridad referencial se especifica mediante la cláusula FOREIGN KEY (clave externa).

Como vimos en la sección 6.2.4, se puede violar una restricción de integridad referencial cuando se insertan o eliminan tuplas, o cuando se modifica un atributo de clave externa. El diseñador del esquema puede especificar la acción que ha de emprenderse si se viola una restricción de integridad referencial al eliminarse una tupla referida o al modificarse un valor de clave primaria referido, anexando una cláusula de *acción disparada por integridad referencial* a una restricción de clave externa. Las opciones incluyen SET NULL (poner nulo), CASCADE (propagar) y SET DEFAULT (poner por omisión). Toda opción debe calificarse con las palabras ON DELETE (al eliminar) o ON UPDATE (al modificar). Ilustramos esto con el ejemplo de la figura 7.1(b). Aquí, el diseñador de la base de datos escogió SET NULL ON DELETE y CASCADE ON UPDATE para la clave externa NSSSUPER de EMPLEADO. Esto significa que si se *elimina* la tupla de un empleado supervisor, el valor de NSSSUPER se *pone en nulo* (es decir, se le asigna el valor NULL) en todas las tuplas de EMPLEADO que hagan referencia a la tupla eliminada. Si el valor de NSS de un empleado supervisor se *modifica* (por ejemplo, porque al introducirlo se tecleó mal), el nuevo valor se *propaga* a NSSSUPER para todas las tuplas de empleado que hagan referencia a la tupla de empleado modificada.

En general, la acción emprendida por el SGBD cuando se especifica SET NULL o SET DEFAULT es la misma para ON DELETE y para ON UPDATE; el valor de los atributos referentes afectados se cambia a NULL en el caso de SET NULL y al valor por omisión especificado en el caso de SET DEFAULT. La acción correspondiente a CASCADE ON DELETE es eliminar todas las tuplas referentes, en tanto que la acción correspondiente a CASCADE ON UPDATE es cambiar el valor

†Las restricciones de clave y de integridad referencial no se incluyeron en las primeras versiones de SQL. En algunas implementaciones, las claves se especificaban implícitamente en el nivel interno mediante la orden CREATE INDEX (crear índice) (véase la Sec. 7.5).

de la clave externa al valor actualizado (nuevo) de la clave primaria en todas las tuplas referentes. Es obligación del diseñador de la base de datos elegir la acción apropiada y especificarla en el DDL. Por regla general, la opción CASCADE es adecuada para relaciones de "vínculo" como TRABAJA_EN, para relaciones que representan atributos multivaluados como LUGARES_DEPTOS, y para relaciones que representan tipos de entidades débiles, como DEPENDIENTE.

La figura 7.1(b) también ilustra la posibilidad de asignar un nombre a una restricción, después de la palabra CONSTRAINT (restricción). Los nombres de todas las restricciones dentro de un esquema en particular deben ser únicos. Los nombres de restricciones sirven para identificar una restricción dada en caso de que se le deba desechar después para sustituirla por otra restricción, como veremos en la sección 7.1.4. La asignación de nombres a las restricciones es opcional.

En la terminología de SQL; las relaciones declaradas mediante instrucciones CREATE TABLE se denominan **tablas base** (o relaciones base). Esto significa que el SGBD crea y almacena realmente la relación y sus tuplas en un archivo. Las relaciones base se distinguen de las **relaciones virtuales**, creadas mediante la instrucción CREATE VIEW (crear vista, véase la Sec. 7.4), las cuales pueden corresponder o no a un archivo físico real. En SQL se considera que los atributos de una tabla base están *ordenados en la secuencia en que se especifican* en la instrucción CREATE TABLE. Sin embargo, se considera que las filas (tuplas) no están ordenadas.

7.1.3 *Las órdenes DROP SCHEMA y DROP TABLE*

Si ya no se necesita un esquema completo, se puede usar la orden DROP SCHEMA (desechar esquema). Hay dos opciones de *forma de desechar*: CASCADE (propagar) y RESTRICT (restringir). Por ejemplo, si queremos eliminar el esquema de base de datos COMPAÑÍA y todas sus tablas, dominios y demás elementos, se utiliza la opción CASCADE como sigue:

DROP SCHEMA COMPAÑÍA **CASCADE**;

Si se elige la opción RESTRICT en lugar de CASCADE, el esquema se desechará sólo si *no contiene elementos*; en caso contrario, no se ejecutará la orden de desechar.

Si ya no se necesita una relación base de un esquema, podemos eliminarla junto con su definición con la orden DROP TABLE (desechar tabla). Por ejemplo, si ya no queremos manejar información sobre los dependientes de los empleados en la base de datos COMPAÑÍA de la figura 6.6, podemos deshacernos de la relación DEPENDIENTE emitiendo la orden

DROP TABLE DEPENDIENTE **CASCADE**;

Si se escoge la opción RESTRICT en lugar de CASCADE, la tabla se desechará sólo si *no se hace referencia a ella* en ninguna restricción (digamos, en las definiciones de clave externa de otra relación) ni en una vista (véase la Sec. 7.4). Con la opción CASCADE, todas las restricciones y vistas que hagan referencia a la tabla se desecharán automáticamente del esquema, junto con la tabla misma.

7.1.4 *La orden ALTER TABLE*

La definición de una tabla base se puede modificar mediante la orden ALTER TABLE (alterar tabla). Las posibles *acciones de alterar tablas* incluyen la adición o eliminación de una columna (atributo), la modificación de la definición de una columna y la adición o eliminación de restricciones de la tabla. Por ejemplo, si queremos añadir a la relación EMPLEADO del esquema

COMPAÑÍA un atributo para mantenernos al tanto de los puestos que tienen los empleados, podemos usar la orden

ALTER TABLE COMPAÑÍA.EMPLEADO **ADD** PUESTO VARCHAR(12);

Faltará introducir un valor para el nuevo atributo PUESTO de cada tupla EMPLEADO. Esto puede hacerse especificando una cláusula DEFAULT o usando la orden UPDATE (modificar, véase la Sec. 7.3). Si no se especifica cláusula por omisión, el nuevo atributo tendrá NULL en todas las tuplas de la relación inmediatamente después de ejecutarse la orden; por tanto, la restricción NOT NULL *no está permitida* en este caso.

Para desechar una columna, se debe elegir CASCADE o RESTRICT como comportamiento de eliminación. Si se elige CASCADE, todas las restricciones y vistas que hagan referencia a la columna se desecharán automáticamente del esquema, junto con la columna en cuestión. Si se escoge RESTRICT, la orden se ejecutará sólo si ninguna vista ni restricción hace referencia a la columna. Por ejemplo, la siguiente orden elimina el atributo DIRECCIÓN de la tabla base EMPLEADO:

ALTER TABLE COMPAÑÍA.EMPLEADO **DROP** DIRECCIÓN **CASCADE**;

También es posible alterar una definición de columna desechando una cláusula por omisión existente o definiendo una nueva cláusula de este tipo. Los siguientes ejemplos ilustran la cláusula mencionada:

ALTER TABLE COMPAÑÍA.DEPARTAMENTO **ALTER** NSSGTE **DROP DEFAULT**;
ALTER TABLE COMPAÑÍA.DEPARTAMENTO **ALTER** NSSGTE **SET DEFAULT** "333445555";

Por último, podemos alterar las restricciones especificadas para una tabla añadiendo o desechando una restricción. Para poder desechar una restricción, es preciso haberle dado un nombre cuando fue especificada. Por ejemplo, si queremos desechar de la relación EMPLEADO la restricción llamada LLASUPEREMP de la figura 7.1(b), escribiremos

ALTER TABLE COMPAÑÍA.EMPLEADO **DROP CONSTRAINT** LLASUPEREMP **CASCADE**;

Una vez hecho esto, podremos redefinir una restricción de reemplazo añadiendo una nueva restricción a la relación, si es necesario. Esto se especifica con la palabra reservada ADD seguida de la nueva restricción, la cual puede tener o no nombre y puede ser cualquiera de los tipos de restricciones de tablas analizados en la sección 7.1.2.

Las subsecciones anteriores proporcionaron un panorama de las órdenes para definir datos en SQL. Existen muchos otros detalles y opciones, y recomendamos al lector interesado consultar los documentos de SQL y SQL2 mencionados en las notas bibliográficas. En la próxima sección analizaremos las capacidades de consulta que tiene SQL.

7.2 Consultas en SQL

SQL tiene una instrucción básica para obtener información de una base de datos: la **instrucción SELECT** (seleccionar). Esta instrucción *no tiene que ver* con la operación SELECCIONAR del álgebra relacional que vimos en el capítulo 6. La instrucción SELECT de SQL tiene muchas opciones y matices, por lo que presentaremos sus características gradualmente. Utilizaremos consultas sencillas especificadas en términos del esquema de la figura 6.5, y nos referiremos al estado de base de datos de muestra que aparece en la figura 6.6 para ilustrar los resultados de algunas de las consultas de ejemplo.

Antes de continuar, debemos señalar una diferencia importante entre SQL y el modelo relacional formal que estudiamos en el capítulo 6: SQL permite que las tablas (relaciones) tengan dos o más tuplas idénticas en todos los valores de sus atributos. Por tanto, en general, una tabla de SQL no es un *conjunto de tuplas* porque los conjuntos no pueden tener dos miembros idénticos; más bien, es un **multiconjunto** (a veces llamado *bolsa*) de tuplas. Algunas relaciones SQL están obligadas a ser conjuntos porque se ha declarado una restricción de clave o porque se ha usado la opción DISTINCT (distintas) con la instrucción SELECT (que describiremos más adelante en esta sección). Conviene tener presente esta distinción al analizar los ejemplos.

7.2.1 *Consultas SQL básicas*

La forma básica de la instrucción SELECT, en ocasiones denominada **transformación** (*mapping*) o **bloque SELECT FROM WHERE,** consta de las tres cláusulas SELECT, FROM (de) y WHERE (donde) y se construye así:

SELECT <lista de atributos>
FROM <lista de tablas>
WHERE <condición>

donde

- <lista de atributos> es una lista de nombres de los atributos cuyos valores va a obtener la consulta.
- <lista de tablas> es una lista de los nombres de las relaciones requeridas para procesar la consulta.
- <condición> es una expresión condicional (booleana) de búsqueda para identificar las tuplas que obtendrá la consulta.

Ahora ilustraremos la instrucción SELECT básica con algunos ejemplos de consultas, mismos que rotularemos con los mismos números de consulta que aparecen en los capítulos 6 y 8, a fin de facilitar las referencias cruzadas.

CONSULTA 0
Obtener la fecha de nacimiento y la dirección del empleado cuyo nombre es 'José B. Silva'. (Ésta es una consulta nueva que no apareció en el capítulo 6.)

```
C0:  SELECT  FECHAN, DIRECCIÓN
     FROM    EMPLEADO
     WHERE   NOMBREP='José' AND INIC='B' AND APELLIDO='Silva'
```

En esta consulta sólo interviene la relación EMPLEADO en la cláusula FROM. La consulta selecciona las tuplas EMPLEADO que satisfagan la condición de la cláusula WHERE y luego proyecta el resultado sobre los atributos FECHAN y DIRECCIÓN listados en la cláusula SELECT. C0 es similar a la expresión del álgebra relacional

$$\pi_{\text{<FECHAN,DIRECCIÓN}}(\sigma_{\text{NOMBREP='José' Y INIC='B' Y APELLIDO='Silva'}}(\text{EMPLEADO}))$$

Por tanto, una consulta SQL simple con un solo nombre de relación en la cláusula FROM es similar a un par de operaciones SELECCIONAR–PROYECTAR del álgebra relacional. La cláusula SELECT de SQL especifica los *atributos de proyección* y la cláusula WHERE especifica la

condición de selección. La única diferencia es que en la consulta SQL podemos obtener tuplas repetidas en el resultado, porque no se impone la restricción de que una relación sea un conjunto. La figura 7.2(a) muestra el resultado de la consulta C0 con la base de datos de la figura 6.6.

CONSULTA 1
Obtener el nombre y la dirección de todos los empleados que pertenecen al departamento 'Investigación'.

```
C1:  SELECT  NOMBREP, APELLIDO, DIRECCIÓN
     FROM    EMPLEADO, DEPARTAMENTO
     WHERE   NOMBRED='Investigación' AND NÚMEROD=ND
```

(a)

FECHAN	DIRECCIÓN
09-ENE-55	Fresnos 731, Higueras, MX

(b)

NOMBREP	APELLIDO	DIRECCIÓN
José	Silva	Fresnos 731, Higueras, MX
Federico	Vizcarra	Valle 638, Higueras, MX
Ramón	Nieto	Espiga 975, Heras, MX
Josefa	Esparza	Rosas 5631, Higueras, MX

(c)

NÚMEROP	NÚMD	APELLIDO	DIRECCIÓN	FECHAN
10	4	Valdés	Bravo 291, Belén, MX	20-JUN-31
30	4	Valdés	Bravo 291, Belén, MX	20-JUN-31

(d)

E.NOMBREP	E.APELLIDO	S.NOMBREP	S.APELLIDO
José	Silva	Federico	Vizcarra
Federico	Vizcarra	Jaime	Botello
Alicia	Zapata	Jazmín	Váldes
Jazmín	Valdés	Jaime	Botello
Ramón	Nieto	Federico	Vizcarra
Josefa	Esparza	Federico	Vizcarra
Ahmed	Jabbar	Jazmín	Valdés

(e)

NSS
123456789
333445555
999887777
987654321
666884444
453453453
987987987
888665555

(f)

NSS	NOMBRED
123456789	Investigación
333445555	Investigación
999887777	Investigación
987654321	Investigación
666884444	Investigación
453453453	Investigación
987987987	Investigación
888665555	Investigación
123456789	Administración
333445555	Administración
999887777	Administración
987654321	Administración
666884444	Administración
453453453	Administración
987987987	Administración
888665555	Administración
123456789	Dirección
333445555	Dirección
999887777	Dirección
987654321	Dirección
666884444	Dirección
453453453	Dirección
987987987	Dirección
888665555	Dirección

(g)

NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
José	B	Silva	123456789	09-ENE-55	Fresnos 731, Higueras, MX	M	30000	333445555	5
Federico	T	Vizcarra	333445555	08-DIC-45	Valle 638, Higueras, MX	M	40000	888665555	5
Ramón	K	Nieto	666884444	15-SEP-52	Espiga 975, Heras, MX	M	38000	333445555	5
Josefa	A	Esparza	453453453	31-JUL-62	Rosas 5631, Higueras, MX	F	25000	333445555	5

Figura 7.2 Resultados de las consultas especificadas sobre la base de datos de la figura 6.6. (a) Resultado de C0. (b) Resultado de C1. (c) Resultado de C2. (d) Resultado de C8. (e) Resultado de C9. (f) Resultado de C10. (g) Resultado de C1C.

La consulta C1 es similar a una secuencia SELECCIONAR–PROYECTAR–REUNIÓN de operaciones del álgebra relacional. Tales consultas suelen recibir el nombre de **consultas de selección–proyección–reunión**. En la cláusula WHERE de C1, la condición NOMBRED = 'Investigación' es una **condición de selección** y corresponde a una operación SELECCIONAR del álgebra relacional. La condición NÚMEROD = ND es una **condición de reunión**, y corresponde a la condición para efectuar una REUNIÓN en el álgebra relacional. El resultado de la consulta 1 puede presentarse como en la figura 7.2(b). En general, es posible especificar cualquier cantidad de condiciones de selección y reunión en una sola consulta SQL. El siguiente ejemplo es una consulta de selección–proyección–reunión con *dos* condiciones de reunión.

CONSULTA 2
Para cada proyecto ubicado en 'Santiago', listar el número del proyecto, el número del departamento controlador y el apellido, la dirección y la fecha de nacimiento del gerente de ese departamento.

```
C2: SELECT  NÚMEROP, NÚMD, APELLIDO, DIRECCIÓN, FECHAN
    FROM    PROYECTO, DEPARTAMENTO, EMPLEADO
    WHERE   NÚMD=NÚMEROD AND NSSGTE=NSS AND
            LUGARP='Santiago'
```

La condición de reunión NÚMD = NÚMEROD relaciona un proyecto con su departamento controlador, en tanto que la condición de reunión NSSGTE = NSS relaciona el departamento controlador con el empleado que lo dirige. El resultado de la consulta C2 se muestra en la figura 7.2(c).

7.2.2 *Manejo de nombres de atributos ambiguos y empleo de seudónimos*

En SQL se puede usar el mismo nombre para dos (o más) atributos siempre que éstos pertenezcan a *diferentes relaciones*. Si tal es el caso, y una consulta hace referencia a dos o más atributos del mismo nombre, es preciso **calificar** el nombre del atributo con el nombre de la relación, a fin de evitar la ambigüedad. Esto se hace *anteponiendo* el nombre de la relación al nombre del atributo y separando los dos con un punto. A fin de ilustrar lo anterior, supongamos que en las figuras 6.5 y 6.6 los atributos ND y APELLIDO de la relación EMPLEADO se llaman NÚMEROD y NOMBRE y que el atributo NOMBRED de DEPARTAMENTO también se llama NOMBRE; entonces, para evitar la ambigüedad, la consulta C1 se reformularía como se muestra en C1A. Debemos calificar los atributos NOMBRE y NÚMEROD en C1A para indicar a cuáles nos referimos, porque en ambas relaciones se usan los mismos nombres:

```
C1A: SELECT  NOMBREP, EMPLEADO.NOMBRE, DIRECCIÓN
     FROM    EMPLEADO, DEPARTAMENTO
     WHERE   DEPARTAMENTO.NOMBRE='Investigación' AND
             DEPARTAMENTO.NÚMEROD=EMPLEADO.NÚMEROD
```

También puede haber ambigüedad en consultas que hagan referencia dos veces a la misma relación, como en el siguiente ejemplo. Esta consulta no apareció en el capítulo 6, así que le daremos el número 8 para distinguirla de las consultas 1 a 7 de la sección 6.7.

CONSULTA 8
Para cada empleado, obtener su nombre de pila y apellido y el nombre de pila y apellido de su supervisor inmediato.


```
C8: SELECT  E.NOMBREP, E.APELLIDO, S.NOMBREP, S.APELLIDO
    FROM    EMPLEADO E, EMPLEADO S
    WHERE   E.NSSSUPER=S.NSS
```

En este caso, podemos declarar los nombres de relación alternativos E y S, llamados **seudónimos**, para la relación EMPLEADO. El seudónimo puede seguir directamente al nombre de la relación, como en C8, o puede ir después de la palabra reservada AS (como); por ejemplo, EMPLEADO AS E. También es posible cambiar los nombres de los atributos de la relación dentro de la consulta asignándoles seudónimos; por ejemplo, si escribimos

```
EMPLEADO AS E(NP, IN, AP, NSS, FN, DIR, SEX, SAL, NSSS, ND)
```

en la cláusula FROM. NP se convierte en seudónimo de NOMBREP, IN de INIC, AP de APELLIDO, y así sucesivamente. En C8, podemos pensar que E y S son dos *copias distintas* de la relación EMPLEADO; la primera, E, representa empleados en el papel de supervisados, y la segunda, S, representa empleados en el papel de supervisores. Luego, podemos aplicar una REUNIÓN a las dos copias. Desde luego, en realidad sólo hay *una* relación EMPLEADO, y la condición de reunión reúne la relación consigo misma encontrando las tuplas que satisfacen la condición de reunión E.NSSSUPER = S.NSS. Observe que éste es un ejemplo de consulta recursiva de un nivel, como las que vimos en la sección 6.6.2. Al igual que en el álgebra relacional, *no podemos* especificar una consulta recursiva general, con un número desconocido de niveles, en un solo enunciado en SQL.

El resultado de la consulta C8 se muestra en la figura 7.2(d). Siempre que se asignen uno o más seudónimos a una relación, podremos usar esos nombres para representar diferentes referencias a esa relación. Esto permite hacer múltiples referencias a la misma relación. Adviértase que, si queremos hacerlo, podemos usar este mecanismo de seudónimos en cualquier consulta SQL, aunque no necesitemos hacer referencia más de una vez a la misma relación. Por ejemplo, podríamos especificar la consulta C1A como se ve en C1B con el único fin de acortar los nombres de relación antepuestos a los nombres de atributos:

```
C1B: SELECT  E.NOMBREP, E.NOMBRE, E.DIRECCIÓN
     FROM    EMPLEADO E, DEPARTAMENTO D
     WHERE   D.NOMBRE='Investigación' AND D.NÚMEROD=E.NÚMEROD
```

7.2.3 *Cláusulas WHERE no especificadas y empleo de '*'*

Aquí examinaremos dos características más de SQL. La *omisión de la cláusula WHERE* indica una selección de tuplas incondicional; por tanto, *todas las tuplas* de la relación especificada en la cláusula FROM son aceptadas y se seleccionan para el resultado de la consulta. Esto equivale a la condición WHERE TRUE, que significa *todas las filas de la tabla*. Si se especifica más de una relación en la cláusula FROM y no hay cláusula WHERE, se selecciona el PRODUCTO CRUZADO —*todas las posibles combinaciones de tuplas*— de esas relaciones. Por ejemplo, la consulta 9 selecciona los NSS de todos los empleados (Fig. 7.2(e)) y la consulta 10 selecciona todas las combinaciones de un NSS de EMPLEADO y un NOMBRED de DEPARTAMENTO (Fig. 7.2(f)).

CONSULTAS 9 Y 10
Seleccionar todos los NSS de EMPLEADO (C9) y todas las combinaciones de NSS de EMPLEADO y NOMBRED de DEPARTAMENTO (C10) de la base de datos.

```
C9:  SELECT  NSS
     FROM    EMPLEADO

C10: SELECT  NSS, NOMBRED
     FROM    EMPLEADO, DEPARTAMENTO
```

Es en extremo importante especificar todas las condiciones de selección y reunión en la cláusula WHERE; si omitimos alguna de esas condiciones, el resultado puede ser una relación incorrecta y de gran tamaño. Observe que C10 es similar a una operación de PRODUCTO CRUZADO seguida de una operación PROYECTAR en el álgebra relacional. Si especificamos todos los atributos de EMPLEADO y DEPARTAMENTO en C10, obtendremos el PRODUCTO CRUZADO.

Si queremos obtener los valores de todos los atributos de las tuplas seleccionadas, no tenemos que listar los nombres de los atributos explícitamente en SQL; basta con especificar un *asterisco* (*), que significa *todos los atributos*. Por ejemplo, la consulta C1C obtiene los valores de todos los atributos de las tuplas de los empleados que pertenecen al departamento 5 (Fig. 7.2(g)); la consulta C1D obtiene todos los atributos de un EMPLEADO y los atributos del DEPARTAMENTO al que pertenece, para todos los empleados del departamento 'Investigación', y C10A especifica el PRODUCTO CRUZADO de las relaciones EMPLEADO y DEPARTAMENTO.

```
C1C:  SELECT  *
      FROM    EMPLEADO
      WHERE   ND=5

C1D:  SELECT  *
      FROM    EMPLEADO, DEPARTAMENTO
      WHERE   NOMBRED='Investigación' AND ND=NÚMEROD

C10A: SELECT  *
      FROM    EMPLEADO, DEPARTAMENTO
```

7.2.4 *Tablas como conjuntos en SQL*

Como ya mencionamos, SQL no trata las relaciones, en general, como conjuntos; *puede haber tuplas repetidas* en una relación o en el resultado de una consulta. SQL no elimina automáticamente las tuplas repetidas en los resultados de las consultas, por las siguientes razones:

- La eliminación de duplicados es una operación costosa. Una forma de implementarla es ordenar las tuplas primero y luego eliminar los duplicados.
- Es posible que el usuario desee ver las tuplas repetidas en el resultado de una consulta.
- Cuando se aplica una función agregada (véase la Sec. 7.2.9) a tuplas, en la mayoría de los casos no querremos eliminar los duplicados.

Si *lo que queremos* es eliminar las tuplas repetidas del resultado de una consulta SQL, nos valdremos de la palabra reservada DISTINCT en la cláusula SELECT, para indicar que sólo deben conservarse tuplas distintas en el resultado. Esto hace que el resultado de una consulta SQL sea una relación —un conjunto de tuplas— de acuerdo con la definición de relación dada en el capítulo 6. Por ejemplo, la consulta 11 obtiene el salario de todos los empleados; si varios de ellos tienen el mismo salario, el valor correspondiente aparecerá varias veces en el resultado de la consulta, como se ilustra en la figura 7.3(a).

CONSULTA 11
Obtener el salario de todos los empleados.

```
C11:  SELECT  SALARIO
      FROM    EMPLEADO
```

Si sólo nos interesan los valores de salario distintos, querremos que dichos valores aparezcan sólo una vez, sin importar cuántos empleados perciban ese salario. Si usamos la palabra reservada DISTINCT como en C11A, lograremos nuestro objetivo; esto se ilustra en la figura 7.3(b):

```
C11A: SELECT  DISTINCT SALARIO
      FROM    EMPLEADO
```

Algunas de las operaciones de conjuntos del álgebra relacional se han incorporado directamente en SQL. Hay una operación de unión de conjuntos (UNION), y en SQL2 hay también operaciones de diferencia de conjuntos (EXCEPT) y de intersección de conjuntos (INTERSECT). Las relaciones resultantes de estas operaciones son conjuntos de tuplas; es decir, *las tuplas repetidas se eliminan del resultado* (a menos que la operación vaya seguida de la palabra reservada ALL (todas)). Como las operaciones de conjuntos sólo se aplican a *relaciones compatibles con la unión*, debemos asegurarnos de que las dos relaciones a las que apliquemos la operación tengan los mismos atributos y de que éstos aparezcan en el mismo orden en ambas relaciones. El siguiente ejemplo ilustra el empleo de UNION.

CONSULTA 4
Preparar una lista con todos los números de los proyectos en los que participa un empleado de apellido 'Silva', sea como trabajador o como gerente del departamento que controla el proyecto.

```
C4:  (SELECT  NÚMEROP
      FROM    PROYECTO, DEPARTAMENTO, EMPLEADO
      WHERE   NÚMD=NÚMEROD AND NSSGTE=NSS AND APELLIDO='Silva')
      UNION
      (SELECT  NÚMEROP
      FROM    PROYECTO, TRABAJA_EN, EMPLEADO
      WHERE   NÚMEROP=NÚMP AND NSSE=NSS AND APELLIDO='Silva')
```

(a)

SALARIO
30000
40000
25000
43000
38000
25000
25000
55000

(b)

SALARIO
30000
40000
25000
43000
38000
55000

(c)

NOMBREP	APELLIDO

(d)

NOMBREP	APELLIDO
Jaime	Botello

Figura 7.3 Resultados de algunas otras consultas especificadas sobre la base de datos de la figura 6.6. (a) Resultado de C11. (b) Resultado de C11A. (c) Resultado de C12. (d) Resultado de C14.

La primera consulta SELECT obtiene los proyectos en los que participa un 'Silva' como gerente del departamento que controla el proyecto, y la segunda obtiene los proyectos en los que participa un 'Silva' como trabajador. Observe que si varios empleados se apellidan 'Silva', se obtendrán los nombres de los proyectos en los que intervenga cualquiera de ellos. La aplicación de la operación UNION a las dos consultas SELECT da el resultado deseado.

7.2.5 *Consultas anidadas y comparaciones de conjuntos*★

En algunas consultas es preciso obtener valores existentes en la base de datos para usarlos en una condición de comparación. Una forma cómoda de formular tales consultas es mediante **consultas anidadas**, que son consultas SELECT completas dentro de la cláusula WHERE de otra consulta, la cual se denomina **consulta exterior**. La consulta 4 se formula en C4 sin una consulta anidada, pero puede reformularse con consultas anidadas, como en C4A:

```
C4A:  SELECT   DISTINCT NÚMEROP
      FROM     PROYECTO
      WHERE    NÚMEROP IN   (SELECT  NÚMEROP
                            FROM    PROYECTO, DEPARTAMENTO,
                                    EMPLEADO
                            WHERE   NÚMD=NÚMEROD AND NSSGTE=NSS
                                    AND APELLIDO='Silva')
               OR
               NÚMEROP IN   (SELECT  NÚMP
                            FROM    TRABAJA_EN, EMPLEADO
                            WHERE   NSSE=NSS AND APELLIDO='Silva')
```

La primera consulta anidada selecciona los números de los proyectos en que un 'Silva' participa como gerente, y la segunda selecciona los números de los proyectos en que un 'Silva' participa como trabajador. En la consulta exterior, seleccionamos una tupla PROYECTO si el valor de NÚMEROP de esa tupla está en el resultado de cualquiera de las dos consultas anidadas. El operador de comparación IN (en) compara un valor v con un conjunto (o multiconjunto) de valores V y produce el valor TRUE (verdadero) si v es uno de los elementos de V.

El operador IN también puede comparar una tupla de valores entre paréntesis con un conjunto de tuplas compatibles con la unión. Por ejemplo, la consulta

```
SELECT DISTINCT NSSE
FROM TRABAJA_EN
WHERE (NÚMP, HORAS) IN (SELECT NÚMP, HORAS FROM TRABAJA_EN
                        WHERE  NSSE='123456789');
```

seleccionará los números de seguro social de todos los empleados que trabajan para la misma combinación (horas, proyecto) en algún proyecto en el que trabaja el empleado 'José Silva' (cuyo NSS = '123456789').

Además del operador IN, podemos usar varios otros operadores de comparación para comparar un valor individual v (por lo regular un nombre de atributo) con un conjunto V (por lo regular una consulta anidada). El operador = ANY (o = SOME) devuelve TRUE si el valor v es igual a *algún valor* del conjunto V, y por tanto equivale a IN. Las palabras reservadas ANY (cualquiera) y SOME (algún) tienen el mismo significado. Otros operadores que se pueden combinar con ANY (o SOME) incluyen >, >=, <, <= y <>. También es posible combinar la palabra reservada ALL (todas) con uno de estos operadores. Por ejemplo, la condición de comparación (v > ALL V) devuelve TRUE si el valor v es mayor que *todos* los valores del conjunto V. Un ejemplo es la siguiente consulta, que devuelve los nombres de los empleados cuyo salario es mayor que el de todos los empleados del departamento 5:

```
SELECT   APELLIDO, NOMBREP
FROM     EMPLEADO
WHERE    SALARIO > ALL (SELECT SALARIO FROM EMPLEADO
                        WHERE      ND=5);
```


En general, podemos tener varios niveles de consultas anidadas. Una vez más, nos enfrentamos a posibles ambigüedades en los nombres de los atributos si hay atributos con el mismo nombre: uno en una relación listada en la cláusula FROM de la *consulta exterior* y el otro en una relación listada en la cláusula FROM de la *consulta anidada*. La regla es que una referencia a un *atributo no calificado* se refiere a la relación declarada en la **consulta anidada más interior**. Por ejemplo, en la cláusula SELECT y en la cláusula WHERE de la primera consulta anidada de C4A, una referencia a cualquier atributo no calificado de la relación PROYECTO se refiere a la relación PROYECTO especificada en la cláusula FROM de la consulta anidada. Si queremos referirnos a un atributo de la relación PROYECTO especificada en la consulta exterior, podemos especificar un *seudónimo* de esa relación y referirnos a él. Estas reglas son similares a las reglas de alcance (validez) para las variables de programa en un lenguaje como PASCAL, que permite anidar procedimientos y funciones. Para ilustrar la ambigüedad potencial de los nombres de atributos en las consultas anidadas, consideremos la consulta 12, cuyo resultado se muestra en la figura 7.3(c).

CONSULTA 12
Obtener el nombre de todos los empleados que tienen un dependiente con el mismo nombre de pila y sexo que el empleado.

```
C12:  SELECT   E.NOMBREP, E.APELLIDO
      FROM     EMPLEADO E
      WHERE    E.NSS IN   (SELECT   NSSE
                           FROM     DEPENDIENTE
                           WHERE    NSSE=E.NSS AND E.NOMBREP
                                    =NOMBRE_ DEPENDIENTE AND
                                    SEXO=E.SEXO)
```

En la consulta anidada de C12, debemos calificar E.SEXO porque se refiere al atributo SEXO del EMPLEADO de la consulta exterior, y DEPENDIENTE también tiene un atributo llamado SEXO. Todas las referencias no calificadas a SEXO en la consulta anidada se refieren a SEXO de DEPENDIENTE. Sin embargo, no *tenemos que* calificar NOMBREP ni NSS porque la relación DEPENDIENTE no tiene atributos llamados NOMBREP ni NSS, así que no hay ambigüedad. Obsérvese que es necesaria la condición NSS = NSSE en la cláusula WHERE de la consulta anidada; sin esta condición, seleccionaríamos los empleados cuyo nombre de pila y sexo coincidieran con los de *cualquier* dependiente, sea que dependiera de ese empleado en particular o no.

Siempre que una condición en la cláusula WHERE de una consulta anidada hace referencia a un atributo de una relación declarada en la consulta exterior, se dice que las dos consultas están **correlacionadas**. Podemos entender mejor qué es una consulta correlacionada si consideramos que la *consulta anidada se evalúa una sola vez para cada tupla (o combinación de tuplas) en la consulta exterior*. Por ejemplo, podemos visualizar C12 como sigue: para *cada* tupla EMPLEADO, evaluar la consulta anidada, que obtiene los valores de NSSE de todas las tuplas DEPENDIENTE con los mismos números de seguro social, sexo y nombre de pila

que la tupla EMPLEADO; si el valor de NSS de la tupla EMPLEADO está *en* el resultado de la consulta anidada, entonces seleccionar esa tupla EMPLEADO.

En general, una consulta escrita con bloques SELECT... FROM... WHERE anidados y que emplee los operadores de comparación = o IN *siempre* puede expresarse como una consulta de un solo bloque. Por ejemplo, C12 puede escribirse como en C12A:

```
C12A:  SELECT   E.NOMBREP, E.APELLIDO
       FROM     EMPLEADO E, DEPENDIENTE D
       WHERE    E.NSS=D.NSSE AND E.SEXO=D.SEXO AND
                E.NOMBREP=D.NOMBRE_DEPENDIENTE
```

La implementación original de SQL en SYSTEM R tenía además un operador de comparación CONTAINS (contiene), que servía para comparar dos conjuntos. Este operador se eliminó posteriormente del lenguaje, tal vez debido a la dificultad para implementarlo de manera eficiente. La mayoría de las implementaciones comerciales de SQL *no* cuentan con dicho operador, el cual compara dos conjuntos de valores y devuelve TRUE si un conjunto contiene todos los valores del otro. La consulta 3 ilustra el empleo del operador CONTAINS.

CONSULTA 3
Obtener el nombre de todos los empleados que trabajan en *todos* los proyectos controlados por el departamento número 5.

```
C3:  SELECT   NOMBREP, APELLIDO
     FROM     EMPLEADO
     WHERE    ((SELECT   NÚMP
               FROM     TRABAJA_EN
               WHERE    NSS=NSSE)
               CONTAINS
               (SELECT  NÚMEROP
               FROM     PROYECTO
               WHERE    NÚMD=5))
```

En C3, la segunda consulta anidada (que no está correlacionada con la exterior) obtiene los números de todos los proyectos controlados por el departamento 5. Para *cada* tupla de empleado, la primera consulta anidada (que sí está correlacionada) obtiene los números de los proyectos en los que trabaja el empleado; si éstos contienen los números de todos los proyectos controlados por el departamento 5, se seleccionará la tupla del empleado y se obtendrá el nombre de ese empleado. Obsérvese que la función del operador de comparación CONTAINS es similar a la operación DIVISIÓN del álgebra relacional, descrita en la sección 6.5.7.

7.2.6 *Las funciones EXISTS y UNIQUE en SQL**

A continuación estudiaremos una función de SQL, llamada EXISTS, que sirve para comprobar si el resultado de una consulta anidada correlacionada está vacío (no contiene tuplas). Ilustraremos el empleo de EXISTS —y también de NOT EXISTS— con algunos ejemplos. Primero, formularemos la consulta C12 de una forma alternativa empleando EXISTS. Esto se ilustra en C12B:


```
C12B:  SELECT   E.NOMBREP, E.APELLIDO
       FROM     EMPLEADO E
       WHERE    EXISTS   (SELECT   *
                         FROM     DEPENDIENTE
                         WHERE    E.NSS=NSSE AND SEXO=E.SEXO
                                  AND E.NOMBREP= NOMBRE_
                                  DEPENDIENTE)
```

EXISTS y NOT EXISTS casi siempre se usan junto con una consulta anidada correlacionada. En C12B, la consulta anidada hace referencia a los atributos NSS, NOMBREP y SEXO de la relación EMPLEADO en la consulta exterior. Podemos visualizar la consulta C12B así: para cada tupla EMPLEADO, evaluar la consulta anidada, que devuelve todas las tuplas DEPENDIENTE con los mismos número de seguro social, sexo y nombre de pila que la tupla EMPLEADO; si existe por lo menos una tupla en el resultado de la consulta anidada, seleccionar esa tupla EMPLEADO. En general, EXISTS(Q) devuelve TRUE si hay *por lo menos una tupla* en el resultado de la consulta Q, y devuelve FALSE en caso contrario; NOT EXISTS(Q) devuelve TRUE si *no hay tuplas* en el resultado de la consulta Q, y devuelve FALSE en caso contrario. A continuación, ilustramos el empleo de NOT EXISTS.

CONSULTA 6
Obtener los nombres de los empleados que no tienen dependientes.

```
C6:  SELECT   NOMBREP, APELLIDO
     FROM     EMPLEADO
     WHERE    NOT EXISTS   (SELECT   *
                           FROM     DEPENDIENTE
                           WHERE    NSS=NSSE)
```

En C6, la consulta anidada correlacionada obtiene todas las tuplas DEPENDIENTE relacionadas con una tupla EMPLEADO. Si *no existe ninguna*, se selecciona la tupla EMPLEADO. Podemos entender mejor la consulta si la visualizamos así: para *cada* tupla EMPLEADO, la consulta anidada selecciona todas las tuplas DEPENDIENTE cuyos valores de NSSE coincidan con el valor de NSS de EMPLEADO; si el resultado de la consulta anidada está vacío, significa que no hay dependientes relacionados con el empleado, así que seleccionamos esa tupla EMPLEADO y devolvemos su NOMBREP y APELLIDO. Existe otra función de SQL, UNIQUE(Q), que devuelve TRUE si no hay tuplas repetidas en el resultado de la consulta Q; en caso contrario, devuelve FALSE.

CONSULTA 7
Listar los nombres de los gerentes que tienen por lo menos un dependiente.

```
C7:  SELECT   NOMBREP, APELLIDO
     FROM     EMPLEADO
     WHERE    EXISTS   (SELECT   *
                        FROM     DEPENDIENTE
                        WHERE    NSS=NSSE)
              AND
              EXISTS   (SELECT   *
                        FROM     DEPARTAMENTO
                        WHERE    NSS=NSSGTE)
```

En C7 se muestra una forma de escribir esta consulta, ahí especificamos dos consultas anidadas correlacionadas: la primera selecciona todas las tuplas DEPENDIENTE relacionadas

con un EMPLEADO, y la segunda selecciona todas las tuplas DEPARTAMENTO dirigidas por el EMPLEADO. Si existen por lo menos una de las primeras y por lo menos una de las segundas, seleccionaremos la tupla EMPLEADO y devolveremos sus valores de NOMBREP y APELLIDO. ¿Puede el lector reescribir esta consulta con una sola consulta anidada o con ninguna?

La consulta 3, con la que ilustramos el operador de comparación CONTAINS, se puede expresar usando EXISTS y NOT EXISTS en los sistemas SQL que *no tengan el operador* CONTAINS. Mostramos la forma de hacerlo en la consulta C3A. Observe que en C3A necesitamos dos niveles de anidamiento y que esta formulación es bastante más compleja que C3, que utilizaba el operador de comparación CONTAINS:

```
C3A:  SELECT  APELLIDO, NOMBREP
      FROM    EMPLEADO
      WHERE   NOT EXISTS
              (SELECT *
              FROM TRABAJA_EN B
              WHERE  (B.NÚMP IN   (SELECT  NÚMEROP
                                   FROM    PROYECTO
                                   WHERE   NÚMD=5))
                     AND
                     NOT EXISTS   (SELECT  *
                                   FROM    TRABAJA_EN C
                                   WHERE   C.NSSE=NSS
                                           AND C.NÚMP=B.NÚMP))
```

En C3A, la consulta anidada exterior selecciona todas las tuplas TRABAJA_EN (B) cuyo NÚMP sea el de un proyecto controlado por el departamento 5, *si es que no* hay una tupla TRABAJA_EN (C) con el mismo NÚMP y con el mismo NSS que la tupla EMPLEADO considerada en la consulta exterior. Si no existe ninguna tupla así, seleccionamos la tupla EMPLEADO. La forma de C3A coincide con la siguiente reformulación de la consulta 3: seleccionar todos los empleados tales que no exista un proyecto controlado por el departamento 5 en el cual no trabaje el empleado.

Cabe señalar que en el álgebra relacional la consulta 3 suele expresarse empleando la operación DIVISIÓN. Además, esta consulta requiere un tipo de cuantificador que en el cálculo relacional se denomina **cuantificador universal** (véase la Sec. 8.1.5). El cuantificador existencial negado NOT EXISTS puede servir para expresar una consulta cuantificada universalmente, como veremos en el capítulo 8.

7.2.7 *Conjuntos explícitos y valores nulos en SQL*★

Ya hemos visto varias consultas con una consulta anidada en la cláusula WHERE. También es posible usar un **conjunto explícito de valores** en dicha cláusula, en vez de una consulta anidada. En SQL, los conjuntos de este tipo se encierran entre paréntesis.

CONSULTA 13
Obtener el número de seguro social de todos los empleados que trabajan en los proyectos 1, 2 o 3.

```
C13:  SELECT  DISTINCT NSSE
      FROM    TRABAJA_EN
      WHERE   NÚMP IN (1, 2, 3)
```


En SQL las consultas pueden comprobar si un valor es NULL, ya sea que falte, no esté definido o no sea aplicable. Sin embargo, en vez de usar = o ≠ para comparar un atributo con NULL, SQL usa IS (es) o IS NOT (no es). La razón es que SQL considera que cada valor nulo es distinto de los demás valores nulos, por lo que la comparación de igualdad no es apropiada. De esto se sigue que, cuando se especifica una condición de búsqueda de reunión (comparación), las tuplas con valores nulos en el atributo de reunión no se incluirán en el resultado (a menos que se trate de una REUNIÓN EXTERNA; véase la sección 7.2.8). La consulta 14 ilustra lo anterior; su resultado se muestra en la figura 7.3(d).

CONSULTA 14
Obtener los nombres de todos los empleados que no tienen supervisores.

```
C14:  SELECT  NOMBREP, APELLIDO
      FROM    EMPLEADO
      WHERE   NSSSUPER IS NULL
```

7.2.8 *Cambio de nombre de los atributos y tablas reunidas*★

Es posible cambiar el nombre de cualquier atributo que aparezca en el resultado de una consulta si se añade el calificador AS seguido del nuevo nombre. Así pues, la construcción AS servirá para declarar seudónimos tanto de atributos como de relaciones. Por ejemplo, la consulta C8A muestra cómo modificar ligeramente C8 para obtener el apellido de cada empleado y de su supervisor, cambiando al mismo tiempo los nombres de los atributos resultantes a NOMBRE_EMPLEADO y NOMBRE_SUPERVISOR. Los nuevos nombres aparecerán en el resultado de la consulta como cabeceras de columnas:

```
C8A:  SELECT  E.APELLIDO AS NOMBRE_EMPLEADO, S.APELLIDO AS
              NOMBRE_SUPERVISOR
      FROM    EMPLEADO AS E, EMPLEADO AS S
      WHERE   E.NSSSUPER=S.NSS
```

El concepto de **tabla reunida** (o **relación reunida**) se incorporó a SQL2 para que los usuarios pudieran especificar una tabla resultante de una operación de reunión *en la cláusula* FROM de una consulta. Quizá sea más fácil comprender esta construcción, en vez de mezclar todas las condiciones de selección y de reunión en la cláusula WHERE. Por ejemplo, consideremos la consulta C1, que obtiene el nombre y la dirección de todos los empleados que trabajan para el departamento de 'Investigación'. Para algunos usuarios, podría ser más fácil especificar primero la reunión de las relaciones EMPLEADO y DEPARTAMENTO, y luego seleccionar las tuplas y atributos deseados. Esto puede escribirse en SQL2 como en C1A:

```
C1A:  SELECT  NOMBREP, APELLIDO, DIRECCIÓN
      FROM    (EMPLEADO JOIN DEPARTAMENTO ON ND=NÚMEROD)
      WHERE   NOMBRED='Investigación'
```

La cláusula FROM de C1A contiene una sola *tabla reunida*. Los atributos de dicha tabla son todos los atributos de la primera tabla, EMPLEADO, seguidos de todos los atributos de la segunda, DEPARTAMENTO. El concepto de tabla reunida también permite al usuario especificar diferentes tipos de reunión, como NATURAL JOIN (reunión natural) y diversos tipos de OUTER JOIN (reunión externa). En una NATURAL JOIN de dos relaciones R y S no se especifica condición de reunión; se crea una condición de (equi)reunión implícita para *cada par de atributos*

con el mismo nombre de R y S. Cada uno de estos pares de atributos sólo se incluye una vez en la relación resultante (véase la Sec 6.5.5).

Si los nombres de los atributos de reunión no son los mismos en las relaciones base, es posible cambiar los nombres para que coincidan y luego aplicar la reunión natural. En este caso se puede usar la construcción AS para cambiar el nombre de una relación y de todos sus atributos. Esto se ilustra en C1B, donde el nombre de la relación DEPARTAMENTO se cambia a DEPTO y los nombres de sus atributos a NOMBRED, ND (para que coincida con el nombre del atributo de reunión deseado ND de EMPLEADO), NSSG y FECHAIG. La condición de reunión implícita para esta reunión natural es EMPLEADO.ND = DEPTO.ND, porque éste es el único par de atributos con el mismo nombre:

```
C1B:  SELECT   NOMBREP, APELLIDO, DIRECCIÓN
      FROM     (EMPLEADO NATURAL JOIN (DEPARTAMENTO AS DEPTO
               (NOMBRED, ND, NSSG, FECHAIG)))
      WHERE    NOMBRED='Investigación'
```

En una tabla reunida, el tipo de reunión por omisión es la reunión **interna**, en la que sólo se incluye una tupla en el resultado si existe una tupla que coincida con una de la otra relación. Por ejemplo, en la consulta C8A, sólo aparecen en el resultado los empleados que *tienen un supervisor*; si una tupla EMPLEADO tiene NULL como valor de NSSSUPER quedará excluida. Si el usuario requiere la inclusión de todos los empleados, deberá usar una reunión externa explícita (véase la Sec. 6.6.3). En SQL2 esto se maneja especificando explícitamente la reunión externa en una tabla reunida, como se ilustra en C8B:

```
C8B:  SELECT   E.APELLIDO AS NOMBRE_EMPLEADO, S.APELLIDO AS
               NOMBRE_SUPERVISOR
      FROM     (EMPLEADO E LEFT OUTER JOIN EMPLEADO S ON
               E.NSSSUPER=S.NSS)
```

Las opciones disponibles para especificar tablas reunidas en SQL2 son INNER JOIN (reunión interna, equivalente a JOIN), LEFT OUTER JOIN (reunión externa izquierda), RIGHT OUTER JOIN (reunión externa derecha) y FULL OUTER JOIN (reunión externa completa). En las últimas tres puede omitirse la palabra OUTER. También es posible *anidar* las especificaciones de reunión; es decir, una de las tablas de una reunión puede ser ella misma una tabla reunida. Esto se ilustra en C2A, que es una forma diferente de especificar la consulta C2, ahora con el concepto de tabla reunida:

```
C2A:  SELECT   NÚMEROP, NÚMD, APELLIDO, DIRECCIÓN, FECHAN
      FROM     ((PROYECTO JOIN DEPARTAMENTO ON NÚMD=NÚMEROD) JOIN
               EMPLEADO ON NSSGTE=NSS)
      WHERE    LUGARP='Santiago'
```

7.2.9 *Funciones agregadas y agrupación*★

En la sección 6.6.1 presentamos el concepto de función agregada como una operación relacional. Como la agrupación y la agregación son necesarios en muchas aplicaciones de bases de datos, SQL cuenta con características que incorporan estos conceptos. La primera de ellas es

una serie de funciones integradas: COUNT (cuenta), SUM (suma), MAX (máximo), MIN (mínimo) y AVG (promedio). La función COUNT devuelve el número de tuplas o valores especificados en una consulta. Las funciones SUM, MAX, MIN y AVG se aplican a un conjunto o multiconjunto de valores numéricos y devuelven, respectivamente, la suma, el valor máximo, el valor mínimo y el promedio (la media) de esos valores. Estas funciones se pueden usar en la cláusula SELECT o en una cláusula HAVING (que pronto presentaremos). Ilustraremos el empleo de estas funciones con algunos ejemplos de consultas.

CONSULTA 15
Obtener la suma de los salarios de todos los empleados, el salario máximo, el salario mínimo y el salario medio.

```
C15:  SELECT   SUM (SALARIO), MAX (SALARIO), MIN (SALARIO),
               AVG (SALARIO)
      FROM     EMPLEADO
```

Si queremos obtener los valores de las funciones anteriores para los empleados de un departamento específico —digamos, el departamento 'Investigación'— podemos escribir la consulta 16, donde la cláusula WHERE restringe las tuplas EMPLEADO a los empleados que trabajan para el departamento 'Investigación'.

CONSULTA 16
Hallar la suma de los salarios de todos los empleados del departamento 'Investigación', así como el salario máximo, el mínimo y el medio en dicho departamento.

```
C16:  SELECT   SUM (SALARIO), MAX (SALARIO), MIN (SALARIO),
               AVG (SALARIO)
      FROM     EMPLEADO, DEPARTAMENTO
      WHERE    ND=NÚMEROD AND NOMBRED='Investigación'
```

CONSULTAS 17 Y 18
Obtener el total de empleados de la compañía (C17) y el número de empleados del departamento 'Investigación' (C18).

```
C17:  SELECT   COUNT (*)
      FROM     EMPLEADO

C18:  SELECT   COUNT (*)
      FROM     EMPLEADO, DEPARTAMENTO
      WHERE    ND=NÚMEROD AND NOMBRED='Investigación'
```

Aquí el asterisco (*) se refiere a las *filas* (tuplas), así que COUNT (*) devuelve el número de filas en el resultado de la consulta. También podemos usar la función COUNT para contar los valores de una columna en vez de las tuplas, como en el siguiente ejemplo.

CONSULTA 19
Contar el número de valores de salario distintos de la base de datos.

```
C19:  SELECT   COUNT (DISTINCT SALARIO)
      FROM     EMPLEADO
```

Hay que ver que, si escribimos COUNT(SALARIO) en vez de COUNT(DISTINCT SALARIO) en C19, obtendremos el mismo resultado que con COUNT(*), porque no se eliminarán los duplicados. Los ejemplos anteriores muestran cómo aplicar funciones para obtener un valor sinóptico de la base de datos. En algunos casos puede ser que necesitemos funciones para seleccionar tuplas específicas. En tales casos especificaremos una consulta anidada correlacionada con la función deseada, y usaremos esa consulta en la cláusula WHERE de una consulta exterior. Por ejemplo, para obtener los nombres de todos los empleados que tienen dos o más dependientes (consulta 5), podemos escribir

```
C5:  SELECT  APELLIDO, NOMBREP
     FROM    EMPLEADO
     WHERE   (SELECT  COUNT (*)
              FROM    DEPENDIENTE
              WHERE   NSS=NSSE) ≥ 2
```

La consulta anidada correlacionada cuenta el número de dependientes que tiene cada empleado; si este número es mayor o igual que 2, se seleccionará la tupla del empleado.

En muchos casos nos interesará aplicar las funciones agregadas a *subgrupos de tuplas de una relación*, con base en los valores de algunos atributos. Tal sería la situación si quisiéramos conocer el salario medio de los empleados de cada departamento o el número de empleados que trabajan en cada proyecto. En estos casos necesitamos agrupar las tuplas que tienen el mismo valor para ciertos atributos, los llamados **atributos de agrupación**, y aplicar la función de manera independiente a cada uno de esos grupos. SQL cuenta con una cláusula GROUP BY (agrupar por) para este fin. La cláusula GROUP BY especifica los atributos de agrupación, que *también deberán aparecer en la cláusula* SELECT, para que el valor que resulte de aplicar cada función a un grupo de tuplas aparezca junto con el valor de los atributos de agrupación.

CONSULTA 20
Para cada departamento, obtener el número de departamento, el número de empleados del departamento y su salario medio.

```
C20:  SELECT    ND, COUNT (*), AVG (SALARIO)
      FROM      EMPLEADO
      GROUP BY  ND
```

En C20 las tuplas EMPLEADO se dividen en grupos, y cada uno de ellos tiene el mismo valor en el atributo de agrupación ND. Las funciones COUNT y AVG se aplican a cada uno de los grupos de tuplas. Observe que la cláusula SELECT sólo incluye el atributo de agrupación y las funciones que han de aplicarse a cada grupo de tuplas. La figura 7.4(a) ilustra el funcionamiento de la agrupación en C20, y también muestra el resultado de ésta.

CONSULTA 21
Para cada proyecto, obtener el número y el nombre del proyecto, así como el número de empleados que trabajan en él.

```
C21:  SELECT    NÚMEROP, NOMBREPR, COUNT (*)
      FROM      PROYECTO, TRABAJA_EN
      WHERE     NÚMEROP=NÚMP
      GROUP BY  NÚMEROP, NOMBREPR
```


(a)

NOMBREP	INIC	APELLIDO	NSS	• • •	SALARIO	NSSSUPER	ND
José	B	Silva	123456789		30000	333445555	5
Federico	T	Vizcarraga	333445555		40000	888665555	5
Ramón	K	Nieto	666884444		38000	333445555	5
Josefa	A	Esparza	453453453	• • •	25000	333445555	5
Alicia	J	Zapata	999887777		25000	987654321	4
Jazmín	S	Valdés	987654321		43000	888665555	4
Ahmed	V	Jabbar	987987987		25000	987654321	4
Jaime	E	Botello	888665555		55000	nulo	1

Agrupación de tuplas EMPLEADO por el valor de ND.

ND	COUNT(*)	AVG (SALARIO)
5	4	33250
4	3	31000
1	1	55000

Resultado de C20.

(b)

NOMBREPR	NÚMEROP		NSSE	NÚMP	HORAS
ProductoX	1		123456789	1	32.5
ProductoX	1		453453453	1	20.0
ProductoY	2		123456789	2	7.5
ProductoY	2		453453453	2	20.0
ProductoY	2		333445555	2	10.0
ProductoZ	3		666884444	3	40.0
ProductoZ	3		333445555	3	10.0
Automatización	10	• • •	333445555	10	10.0
Automatización	10		999887777	10	10.0
Automatización	10		987987987	10	35.0
Reorganización	20		333445555	20	10.0
Reorganización	20		987654321	20	15.0
Reorganización	20		888665555	20	nulo
Nuevasprestaciones	30		987987987	30	5.0
Nuevasprestaciones	30		987654321	30	20.0
Nuevasprestaciones	30		999887777	30	30.0

La condición HAVING de C22 no selecciona estos grupos.

Después de aplicar la cláusula WHERE pero antes de aplicar HAVING.

NOMBREPR	NÚMEROP		NSSE	NÚMP	HORAS
ProductoY	2		123456789	2	7.5
ProductoY	2		453453453	2	20.0
ProductoY	2		333445555	2	10.0
Automatización	10	• • •	333445555	10	10.0
Automatización	10		999887777	10	10.0
Automatización	10		987987987	10	35.0
Reorganización	20		333445555	20	10.0
Reorganización	20		987654321	20	15.0
Reorganización	20		888665555	20	nulo
Nuevasprestaciones	30		987987987	30	5.0
Nuevasprestaciones	30		987654321	30	20.0
Nuevasprestaciones	30		999887777	30	30.0

Después de aplicar la condición de la cláusula HAVING.

NOMBREPR	COUNT (*)
ProductoY	3
Automatización	3
Reorganización	3
Nuevasprestaciones	3

Resultado de C22 (no se muestra NÚMEROP).

Figura 7.4 Resultados de GROUP BY y HAVING. (a) Resultados de la consulta 20. (b) Resultados de la consulta 22.

C21 muestra cómo podemos usar una condición de reunión junto con GROUP BY. En este caso, la agrupación y las funciones se aplican *después* de la reunión de las dos relaciones. Habrá veces en que querremos obtener los valores de estas funciones sólo para *grupos*

que satisfagan ciertas condiciones. Por ejemplo, suponga que se desea modificar la consulta 21 de modo que sólo aparezcan en el resultado los proyectos que tengan más de dos empleados. Para este fin SQL cuenta con una **cláusula HAVING** (que tiene), la cual puede aparecer junto con la cláusula GROUP BY. HAVING especifica una condición en términos del grupo de tuplas asociado a cada valor de los atributos de agrupación; sólo los grupos que satisfagan la condición entrarán en el resultado de la consulta. Esto se ilustra con la consulta 22.

CONSULTA 22
Para cada proyecto *en el que trabajen más de dos empleados*, obtener el número y el nombre del proyecto, así como el número de empleados que trabajan en él.

```
C22:  SELECT     NÚMEROP, NOMBREPR, COUNT (*)
      FROM       PROYECTO, TRABAJA_EN
      WHERE      NÚMEROP=NÚMP
      GROUP BY   NÚMEROP, NOMBREPR
      HAVING     COUNT (*) > 2
```

Adviértase que, en tanto las condiciones de selección de la cláusula WHERE limitan las *tuplas* a las que se aplican las funciones, la cláusula HAVING limita *grupos enteros*. La figura 7.4(b) ilustra el empleo de HAVING y presenta el resultado de C22.

CONSULTA 23
Para cada proyecto, obtener el número y el nombre del proyecto, así como el número de empleados del departamento 5 que trabajan en el proyecto.

```
C23:  SELECT     NÚMEROP, NOMBREPR, COUNT (*)
      FROM       PROYECTO, TRABAJA_EN, EMPLEADO
      WHERE      NÚMEROP=NÚMP AND NSS=NSSE AND ND=5
      GROUP BY   NÚMEROP, NOMBREPR
```

Aquí restringimos las tuplas de cada grupo a las que satisfacen la condición especificada en la cláusula WHERE; a saber, que trabajan en el departamento número 5. Observe que debemos tener especial cuidado cuando se aplican dos condiciones diferentes (una a la función de la cláusula SELECT y otra a la función de la cláusula HAVING). Por ejemplo, suponga que se desea contar el número *total* de empleados cuyos salarios rebasan los $40 000 en cada departamento, pero sólo en el caso de departamentos en los que trabajen más de cinco empleados. Aquí, la condición (SALARIO > 40000) se aplica sólo a la función COUNT de la cláusula SELECT. Suponga que escribimos la siguiente consulta *incorrecta*:

```
SELECT     NOMBRED, COUNT (*)
FROM       DEPARTAMENTO, EMPLEADO
WHERE      NÚMEROD=ND AND SALARIO>40000
GROUP BY   NOMBRED
HAVING     COUNT (*) > 5
```

Esto no es correcto porque seleccionará sólo los departamentos que tengan más de cinco empleados *que ganen más de* $40 000. La regla es que la cláusula WHERE se ejecuta primero, para seleccionar tuplas individuales; la cláusula HAVING se aplica después, para seleccionar grupos individuales de tuplas. Por tanto, las tuplas ya están restringidas a los empleados que ganan más de $40 000 *antes* de aplicarse la función de la cláusula HAVING. Una forma de escribir la consulta correctamente es con una consulta anidada, como en la consulta 24.

CONSULTA 24
Para cada departamento que tenga más de cinco empleados, obtener el nombre del departamento y el número de empleados que ganan más de $40 000.

```
C24:  SELECT     NOMBRED, COUNT (*)
      FROM       DEPARTAMENTO, EMPLEADO
      WHERE      NÚMEROD=ND AND SALARIO>40000 AND
                 ND IN     (SELECT     ND
                            FROM       EMPLEADO
                            GROUP BY   ND
                            HAVING     COUNT (*) > 5)
      GROUP BY   NOMBRED
```

7.2.10 *Comparaciones de subcadenas, operadores aritméticos y ordenación*

En esta sección estudiaremos otras tres características que tiene el SQL. La primera permite especificar condiciones de comparación de partes de una cadena de caracteres, empleando el operador de comparación **LIKE** (como). Las cadenas parciales se especifican mediante dos caracteres reservados: '%' sustituye a un número arbitrario de caracteres, y '_' sustituye a un solo carácter arbitrario. Por ejemplo, consideremos la siguiente consulta.

CONSULTA 25
Obtener todos los empleados cuya dirección esté en Higueras, Estado de México.

```
C25:  SELECT     NOMBREP, APELLIDO
      FROM       EMPLEADO
      WHERE      DIRECCIÓN LIKE '%Higueras, MX%'
```

Para obtener todos los empleados que nacieron en la década de 1950, podemos usar la consulta 26. Aquí, '5' debe ser el tercer carácter de la cadena (según nuestro formato de fecha), así que usamos el valor '__5_______', donde cada carácter de subrayado sirve como sustituto de un carácter arbitrario.

CONSULTA 26
Encontrar todos los empleados que nacieron en la década de 1950.

```
C26:  SELECT     NOMBREP, APELLIDO
      FROM       EMPLEADO
      WHERE      FECHAN LIKE  '__5_______'
```

Otra característica es la posibilidad de usar aritmética en las consultas. Los operadores aritméticos estándar '+', '–', '*' y '/' (para sumar, restar, multiplicar y dividir, respectivamente) se pueden aplicar a valores numéricos en una consulta. Por ejemplo, suponga que deseamos conocer el efecto de otorgar un aumento del 10% a todos los empleados que trabajan en el proyecto 'ProductoX'; podemos emitir la consulta 27 para ver cuáles serían sus salarios.

CONSULTA 27
Mostrar los salarios resultantes si cada empleado que trabaja en el proyecto 'ProductoX' recibe un aumento del 10%.

```
C27:  SELECT   NOMBREP, APELLIDO, 1.1*SALARIO
      FROM     EMPLEADO, TRABAJA_EN, PROYECTO
      WHERE    NSS=NSSE AND NÚMP=NÚMEROP AND NOMBREPR
               ='ProductoX'
```

En el caso de tipos de datos de cadena, podemos usar el operador de concatenación '||' en una consulta para anexar un valor de cadena a otro. En el caso de tipos de datos de fecha, hora, marca de tiempo e intervalo, los operadores incluyen el incremento ('+') o decremento ('–') de una fecha, hora o marca de tiempo en un intervalo compatible con el tipo. Además podemos especificar un valor de intervalo como la diferencia entre dos valores de fecha, hora o marca de tiempo.

Por último, SQL permite al usuario ordenar las tuplas del resultado de una consulta según los valores de uno o más atributos, empleando la cláusula **ORDER BY** (ordenar por). Por ejemplo, suponga que deseamos obtener una lista de los empleados y los proyectos en los que trabajan, pero queremos que esté ordenada según los departamentos a los que pertenecen los empleados, con los nombres de éstos ordenados alfabéticamente dentro de cada departamento:

CONSULTA 28
Obtener una lista de empleados y de los proyectos en los que trabajan, ordenados por departamento y, dentro de cada departamento, alfabéticamente por apellido y nombre.

```
C28:  SELECT     NOMBRED, APELLIDO, NOMBREP, NOMBREPR
      FROM       DEPARTAMENTO, EMPLEADO, TRABAJA_EN, PROYECTO
      WHERE      NÚMEROD=ND AND NSS=NSSE AND NÚMP=NÚMEROP
      ORDER BY   NOMBRED, APELLIDO, NOMBREP
```

El orden por omisión es el ascendente. Podemos incluir la palabra reservada DESC si queremos que los valores queden en orden descendente. La palabra reservada ASC sirve para especificar explícitamente el orden ascendente. Si queremos que nuestra lista esté en orden descendente por NOMBRED y en orden ascendente por APELLIDO, NOMBREP, la cláusula ORDER BY de C28 se convertirá en

```
ORDER BY  NOMBRED DESC, APELLIDO ASC, NOMBREP ASC
```

7.2.11 *Análisis*

Una consulta en SQL puede constar de un máximo de seis cláusulas, pero sólo son obligatorias las dos primeras, SELECT y FROM. Las cláusulas se especifican en el siguiente orden (las que están entre corchetes son opcionales):

```
SELECT <lista de atributos>
FROM <lista de tablas>
[WHERE <condición>]
[GROUP BY <atributo(s) de agrupación>]
[HAVING <condición de agrupación>]
[ORDER BY <lista de atributos>]
```

La cláusula SELECT indica qué atributos o funciones se van a obtener. La cláusula FROM especifica todas las relaciones que se necesitan en la consulta, incluidas las relaciones reunidas, pero no las que se requieren en consultas anidadas. La cláusula WHERE especifica las condiciones para seleccionar tuplas de esas relaciones. GROUP BY especifica atributos de agrupación, en tanto que HAVING especifica una condición que deben cumplir los grupos seleccionados, no las tuplas individuales. Las funciones agregadas integradas COUNT, SUM, MIN, MAX y AVG se usan junto con la agrupación. Por último, ORDER BY especifica un orden para presentar el resultado de una consulta.

Las consultas se evalúan aplicando primero la cláusula FROM, seguida de la cláusula WHERE, y luego GROUP BY y HAVING. Si no se especifica ninguna de las últimas tres cláusulas, (GROUP BY, HAVING, ORDER BY), podemos *considerar* que la consulta se ejecuta de la siguiente manera: para *cada combinación de tuplas* —una de cada una de las relaciones especificadas en la cláusula FROM— evaluar la cláusula WHERE; si el resultado es TRUE, colocar en el resultado de la consulta los valores de los atributos de esta combinación de tuplas que están especificados en la cláusula SELECT. Desde luego, ésta no es una forma eficiente de implementar la consulta, y cada SGBD tiene rutinas especiales de optimización de consultas para elegir un plan de ejecución. En el capítulo 16 estudiaremos el procesamiento y la optimización de consultas.

En general, hay muchas maneras de especificar la misma consulta en SQL. Esta flexibilidad ofrece ventajas y desventajas. La principal ventaja es que el usuario puede escoger la técnica que le resulte más cómoda para especificar una consulta. Por ejemplo, muchas consultas pueden especificarse con condiciones de reunión en la cláusula WHERE, o con relaciones reunidas en la cláusula FROM, o con alguna forma de consultas anidadas y el operador de comparación IN. Algunos usuarios tal vez se sientan más cómodos con un enfoque, y otros quizá prefieran uno distinto. Desde el punto de vista del programador y de la optimización de consultas del sistema, en general es preferible escribir las consultas con el mínimo de anidamiento y de ordenamiento implícito que sea posible.

La desventaja de contar con muchas formas de especificar la misma consulta es que ello puede confundir al usuario, quien tal vez no sabrá cuál técnica usar para especificar ciertos tipos de consultas.

Otro problema es que puede ser más eficiente ejecutar una consulta especificada de una manera que ejecutar la misma consulta especificada de otro modo. En condiciones ideales, esto no debe ocurrir: el SGBD deberá procesar la misma consulta de la misma manera, sin importar cómo se haya especificado. Sin embargo, en la práctica esto es muy difícil, pues cada SGBD tiene distintos métodos para procesar consultas especificadas de diferentes maneras. Una carga adicional para el usuario es tener que determinar cuál de las especificaciones alternativas es la más eficiente. En condiciones ideales, el usuario sólo debería preocuparse por especificar la consulta correctamente; toca al SGBD ejecutarla de manera eficiente. En la práctica, empero, es mejor que el usuario esté consciente de cuáles tipos de construcciones de consulta tienen un costo de procesamiento más elevado que otros. Por ejemplo, una condición de reunión especificada en términos de campos para los cuales no existen índices (véase la Sec. 7.5) puede ser bastante costosa si se especifica para dos relaciones grandes; por tanto, el usuario debería crear los índices apropiados *antes* de especificar una consulta así.

7.3 Instrucciones de actualización en SQL

Podemos usar tres órdenes en SQL para actualizar la base de datos: INSERT (insertar), DELETE (eliminar) y UPDATE (modificar). Analizaremos estas órdenes una por una.

7.3.1 *La orden* INSERT

En su forma más simple, INSERT sirve para añadir una sola tupla a una relación. Debemos especificar el nombre de la relación y una lista de valores para la tupla. Los valores deberán listarse en el *mismo orden* en que se especificaron los atributos correspondientes en la instrucción CREATE TABLE. Por ejemplo, si queremos añadir una nueva tupla a la relación EMPLEADO que se muestra en la figura 6.5 y que se especificó en la instrucción CREATE TABLE EMPLEADO de la figura 7.1, podemos usar A1:

```
A1:  INSERT INTO   EMPLEADO
     VALUES        ('Ricardo', 'C', 'Martínez', '653298653', '30-DIC-52',
                    'Olmo 98, Cedros, MX', 'M', 37000, '987654321', 4)
```

Una segunda forma de la instrucción INSERT permite al usuario especificar explícitamente nombres de atributos que correspondan a los valores de la orden INSERT. En este caso, los atributos con valores NULL o DEFAULT se pueden *omitir*. Por ejemplo, si queremos introducir una tupla para un nuevo empleado del cual sólo conocemos los atributos NOMBREP, APELLIDO y NSS, podemos usar A1A:

```
A1A: INSERT INTO   EMPLEADO (NOMBREP, APELLIDO, NSS)
     VALUES        ('Ricardo', 'Martínez', '653298653')
```

Los atributos no especificados en A1A reciben el valor por omisión o NULL, y los valores se listan en el mismo orden *en que aparecen los atributos en la instrucción* INSERT *misma*. También es posible insertar en una relación *varias tuplas* separadas por comas en una sola instrucción INSERT. Los valores de atributos que forman cada tupla se encierran entre paréntesis.

Un SGBD con una implementación total de SQL2 deberá manejar e imponer todas las restricciones de integridad que se pueden especificar en el DDL. Sin embargo, algunos SGBD no manejan todas las restricciones, con el fin de mantener la eficiencia del SGBD o debido a la complejidad de imponerlas todas. En un sistema que no maneje alguna restricción —digamos, la integridad referencial— los usuarios o programadores deberán imponerla. Por ejemplo, si emitimos la orden de A2 con la base de datos de la figura 6.6, un SGBD que no maneje la integridad referencial efectuará la inserción aunque no exista en la base de datos ninguna tupla DEPARTAMENTO con NÚMEROD = 2. Es el usuario el que debe asegurarse de que no se violen tales restricciones no verificadas. Sin embargo, el SGBD debe implementar comprobaciones para hacer cumplir todas las restricciones de integridad de SQL *que sí maneje*. Un SGBD que imponga NOT NULL rechazará una orden INSERT en la que un atributo declarado NOT NULL carezca de un valor; por ejemplo, A2A sería *rechazada* porque no se proporciona un valor de NSS:

```
A2:  INSERT INTO   EMPLEADO (NOMBREP, APELLIDO, NSS, ND)
     VALUES        ('Roberto', 'Huerta', '980760540', 2)

A2A: INSERT INTO   EMPLEADO (NOMBREP, APELLIDO, ND) (* actualización
                   rechazada *)
     VALUES        ('Roberto', 'Huerta', 2)
```

Una variación de la instrucción INSERT inserta múltiples tuplas en una relación al tiempo que crea la relación y la carga con el *resultado de una consulta*. Por ejemplo, para crear una tabla temporal con los nombres y el número de los empleados, así como el total de salarios de cada departamento, escribimos las instrucciones de A3A y A3B:


```
A3A: CREATE TABLE  INFO_DEPTOS (NOMBRE_DEPTO VARCHAR(15),
                                NÚM_DE_EMPS  INTEGER,
                                SAL_TOTAL    INTEGER);

A3B: INSERT INTO   INFO_DEPTOS (NOMBRE_DEPTO, NÚM_DE_EMPS,
                   SAL_TOTAL)
                   SELECT    NOMBRED, COUNT (*), SUM (SALARIO)
                   FROM      DEPARTAMENTO, EMPLEADO
                   WHERE     NÚMEROD=ND
                   GROUP BY  NOMBRED;
```

A3A creará la tabla INFO_DEPTOS, y la consulta de A3B la cargará con la información sinóptica obtenida de la base de datos. Ahora podemos consultar INFO_DEPTOS como se hace con cualquier relación; y cuando no la necesitemos más, podremos eliminarla con la orden DROP TABLE. Subrayemos que es posible que la tabla INFO_DEPTOS no esté actualizada; esto es, si actualizamos alguna de las relaciones DEPARTAMENTO o EMPLEADO después de emitir A3B, la información de INFO_DEPTOS *perderá actualidad*. Necesitaremos crear una vista (véase la Sec. 7.4) para que una tabla como ésta se mantenga actualizada.

7.3.2 *La orden* DELETE

La orden DELETE elimina tuplas de una relación. Cuenta con una cláusula WHERE, similar a la de las consultas SQL, para seleccionar las tuplas que se van a eliminar. Las tuplas se eliminan explícitamente de una sola tabla a la vez. Sin embargo, la eliminación puede propagarse a tuplas de otras relaciones si tal acción se especifica en las restricciones de integridad referencial del DDL (véase la Sec. 7.1.2). Dependiendo del número de tuplas seleccionadas por la condición de la cláusula WHERE, una sola orden DELETE puede eliminar cero, una o varias tuplas. La omisión de la cláusula WHERE indica que se deben eliminar todas las tuplas de la relación, aunque la tabla permanecerá en la base de datos como una tabla vacía. Para eliminar del todo dicha tabla tendremos que usar la orden DROP TABLE. Las órdenes DELETE de A4A a A4D, si se aplican independientemente a la base de datos de la figura 6.6, eliminarán cero, una, cuatro y todas las tuplas, respectivamente, de la relación EMPLEADO:

```
A4A: DELETE FROM   EMPLEADO
     WHERE         APELLIDO='Bojórquez'

A4B: DELETE FROM   EMPLEADO
     WHERE         NSS='123456789'

A4C: DELETE FROM   EMPLEADO
     WHERE         ND IN     (SELECT   NÚMEROD
                             FROM      DEPARTAMENTO
                             WHERE     NOMBRED='Investigación')

A4D: DELETE FROM   EMPLEADO
```

7.3.3 *La orden* UPDATE

La orden UPDATE sirve para modificar los valores de los atributos en una o más tuplas seleccionadas. Al igual que en la instrucción DELETE, una cláusula WHERE selecciona de una sola relación las tuplas que se van a modificar. Sin embargo, la actualización de un valor de

clave primaria puede propagarse a los valores de clave externa de tuplas de otras relaciones si tal acción se especifica en las restricciones de integridad referencial del DDL (véase la Sec. 7.1.2). Una **cláusula SET** adicional especifica los atributos que se modificarán y sus nuevos valores. Por ejemplo, si queremos cambiar el lugar y el número del departamento controlador del proyecto número 10 a 'Belén' y 5, respectivamente, usaremos A5:

```
A5:  UPDATE   PROYECTO
     SET      LUGARP = 'Belén', NÚMD = 5
     WHERE    NÚMEROP = 10
```

Es posible modificar varias tuplas con una sola orden UPDATE. Un ejemplo sería otorgar a todos los empleados del departamento 'Investigación' un aumento salarial del 10%, como se muestra en A6. En esta solicitud, el valor modificado de SALARIO depende del valor original de SALARIO en cada tupla, así que se necesitarán dos referencias al atributo SALARIO. En A6, la referencia al atributo SALARIO de la derecha se refiere al valor antiguo de SALARIO, *antes de la modificación*, y el de la izquierda se refiere al nuevo valor de SALARIO, *después de la modificación*:

```
A6:  UPDATE   EMPLEADO
     SET      SALARIO = SALARIO *1.1
     WHERE    ND IN        (SELECT   NÚMEROD
                            FROM     DEPARTAMENTO
                            WHERE    NOMBRED='Investigación')
```

También es posible especificar NULL o DEFAULT como nuevo valor del atributo. Adviértase que cada orden UPDATE especifica explícitamente sólo una relación. Si queremos modificar varias relaciones, deberemos emitir varias órdenes UPDATE. Éstas (y otras instrucciones de SQL) se pueden incorporar en un programa de aplicación general, como se verá en la sección 7.7.

7.4 Vistas en SQL

En esta sección presentaremos el concepto de vista en SQL. Después veremos cómo se especifican las vistas y estudiaremos el problema de cómo actualizar una vista.

7.4.1 *Concepto de vista en SQL*

En la terminología de SQL, una **vista** es una tabla derivada de otras tablas.† Estas otras pueden ser tablas base o vistas previamente definidas. Las vistas no necesariamente existen en forma física: se les considera **tablas virtuales**, en contraste con las tablas base cuyas tuplas se almacenan realmente en la base de datos. Esto limita las operaciones de actualización que es posible aplicar a las vistas, pero no implica limitaciones para consultar estas últimas.

Podemos considerar una vista como una forma de especificar una tabla a la que tendremos que referirnos con frecuencia, aunque tal vez no posea existencia física. Por ejemplo, con el esquema de la figura 6.5, es posible que emitamos consultas con frecuencia para obtener el nombre de un empleado y los nombres de los proyectos en los que trabaja. En vez

†Tal como lo usamos aquí, el término *vista* es más limitado que el término *vistas de usuario* que estudiamos en los capítulos 1 y 2.

de tener que especificar la reunión de las tablas EMPLEADO, TRABAJA_EN y PROYECTO cada vez que se emite una consulta así, podemos definir una vista que sea el resultado de dichas reuniones y que, por tanto, incluya los atributos que deseamos obtener con frecuencia. Así, podremos emitir consultas de esa vista, especificadas como obtenciones de una sola tabla, en vez de obtenciones que impliquen dos reuniones de tres tablas. Llamamos a las tablas EMPLEADO, TRABAJA_EN y PROYECTO las **tablas de definición** de la vista.

7.4.2 *Especificación de vistas en SQL*

La orden para especificar una vista es **CREATE VIEW** (crear vista). La vista recibe un nombre de tabla (virtual), una lista de nombres de atributos y una consulta para especificar el contenido de la vista. Si ninguno de los atributos de la vista es el resultado de aplicar funciones u operaciones aritméticas, no tendremos que especificar nombres de atributos para la vista, pues serán los mismos que los nombres de los atributos de las tablas de definición. Las vistas de V1 y V2 crean tablas virtuales cuyos esquemas se ilustran en la figura 7.5 en términos del esquema de base de datos que se muestra en la figura 6.5

```
V1:  CREATE VIEW   TRABAJA_EN1
     AS  SELECT    NOMBREP, APELLIDO, NOMBREPR, HORAS
         FROM      EMPLEADO, PROYECTO, TRABAJA_EN
         WHERE     NSS=NSSE AND NÚMP=NÚMEROP;

V2:  CREATE VIEW   INFO_DEPTO (NOMBRE_DEPTO, NÚM_DE_EMPS,
                   SAL_TOTAL)
     AS  SELECT      NOMBRED, COUNT (*), SUM (SALARIO)
         FROM        DEPARTAMENTO, EMPLEADO
         WHERE       NÚMEROD=ND
         GROUP BY    NOMBRED;
```

En V1 no especificamos nuevos nombres de atributos para la vista TRABAJA_EN1 (aunque podríamos haberlo hecho); en este caso, TRABAJA_EN1 hereda los nombres que tienen los atributos de la vista de las tablas de definición EMPLEADO, PROYECTO y TRABAJA_EN. La vista V2 especifica explícitamente nuevos nombres de atributos para la vista INFO_DEPTO, por medio de una correspondencia uno a uno entre los atributos especificados en la cláusula CREATE VIEW y los especificados en la cláusula SELECT de la consulta que define la vista. Podemos especificar consultas SQL en términos de una vista —o tabla virtual— de la misma manera que especificamos consultas en términos de tablas base. Por ejemplo, si queremos obtener el apellido y el nombre de pila de todos los empleados que trabajan en el 'ProyectoX', podemos usar la vista TRABAJA_EN1 y especificar la consulta como en CV1:

TRABAJA_EN1

NOMBREP	APELLIDO	NOMBREPR	HORAS

INFO_DEPTO

NOMBRE_DEPTO	NÚM_DE_EMPS	SAL_TOTAL

Figura 7.5 Dos vistas especificadas sobre el esquema de base de datos de la figura 6.5.

```
CV1:  SELECT  NOMBREPR, NOMBREP, APELLIDO
      FROM    TRABAJA_EN1
      WHERE   NOMBREPR='ProyectoX';
```

La misma consulta requeriría la especificación de dos reuniones si se hiciera sobre las relaciones base; la principal ventaja de las vistas es que simplifican la especificación de ciertas consultas. Las vistas también pueden servir como mecanismos de seguridad (véase el Cap. 20).

Las vistas *siempre están actualizadas*; si modificamos las tuplas de las tablas base sobre las que se define la vista, ésta reflejará automáticamente los cambios. Por tanto, una vista no se crea en el momento de definirla, sino más bien en el momento en que se especifica una consulta de la vista. Es responsabilidad del SGBD, y no del usuario, asegurarse de que la vista esté actualizada.

Si ya no necesitamos una vista, podemos usar la orden **DROP VIEW** (desechar vista) para deshacernos de ella. Por ejemplo, si queremos eliminar las dos vistas definidas en V1 y V2, podemos usar las instrucciones de SQL de V1A y V2A:

```
V1A:  DROP VIEW  TRABAJA_EN1;

V2A:  DROP VIEW  INFO_DEPTO;
```

7.4.3 *Actualización de vistas e implementación de vistas*

La actualización de vistas es complicada y puede ser ambigua. En general, una actualización de una vista definida en términos de *una sola tabla base* sin *funciones agregadas* se puede traducir a una actualización de la tabla base subyacente. En el caso de una vista que implique reuniones, la operación de actualización puede traducirse a operaciones de actualización de las relaciones base subyacentes *de varias maneras*. El tema de la actualización de vistas es todavía un área de investigación activa. A fin de ilustrar los problemas potenciales de actualizar una vista definida sobre múltiples tablas, consideremos la vista TRABAJA_EN1 y supongamos que emitimos la orden de modificar el atributo NOMBREPR de 'José Silva', cambiándolo de 'ProductoX' a 'ProductoY'. Esta modificación de vista se muestra en AV1:

```
AV1:  UPDATE  TRABAJA_EN1
      SET     NOMBREPR = 'ProductoY'
      WHERE   APELLIDO='Silva' AND NOMBREP='José' AND NOMBREPR=
              'ProductoX'
```

Esta consulta puede traducirse a varias modificaciones de las relaciones base para lograr la modificación deseada de la vista. En seguida mostramos dos posibles modificaciones de las relaciones base, (a) y (b), que corresponden a AV1:

```
(a):  UPDATE  TRABAJA_EN
      SET     NÚMP = (SELECT NÚMEROP FROM PROYECTO
                      WHERE  NOMBREPR='ProductoY')
      WHERE   NSSE =  (SELECT NSS FROM EMPLEADO
                      WHERE APELLIDO='Silva' AND NOMBREP='José')
                      AND
              NÚMP = (SELECT NÚMEROP FROM PROYECTO
                      WHERE NOMBREPR='ProductoX')
```



```
(b):  UPDATE  PROYECTO
      SET     NOMBREPR = 'ProductoY'
      WHERE   NOMBREPR = 'ProductoX'
```

La modificación (a) relaciona a 'José Silva' con la tupla 'ProductoY' de PROYECTO, y no con la tupla 'ProductoX', lo que es la modificación deseada más probable. Sin embargo, (b) también produciría el efecto deseado sobre la vista, aunque lo haría modificando el nombre de la tupla 'ProductoX' de la relación PROYECTO, cambiándolo a 'ProductoY'. Es muy poco probable que el usuario que especificó la modificación de vista AV1 quiera que la actualización se interprete como en (b), pues tiene el efecto adicional de modificar todas las tuplas de la vista en las que NOMBREPR = 'ProductoX'.

Algunas modificaciones de vistas pueden carecer de sentido; por ejemplo, la modificación del atributo SAL_TOTAL de INFO_DEPTO no tiene sentido porque SAL_TOTAL se define como la suma de los salarios individuales de los empleados. Esta solicitud se muestra como AV2:

```
AV2:  UPDATE  INFO_DEPTO
      SET     SAL_TOTAL=100000
      WHERE   NOMBRED='Investigación';
```

Muchas posibles modificaciones de las relaciones base subyacentes pueden satisfacer esta actualización de vista.

En general, no podemos garantizar que toda vista se pueda actualizar. Una actualización de vista es factible cuando sólo *una posible actualización* de las relaciones base puede lograr el efecto deseado sobre la vista. Siempre que una actualización de la vista pueda traducirse a *más de una actualización* de las relaciones base subyacentes, tendremos que contar con un procedimiento específico para elegir la actualización deseada. Algunos investigadores han creado métodos para escoger la actualización más probable, en tanto que otros prefieren dejar que el usuario elija la traducción de la actualización deseada durante la definición de la vista.

En síntesis, cabe hacer las siguientes observaciones:

- Una vista con una sola tabla de definición es actualizable si los atributos de la vista contienen la clave primaria o alguna otra clave candidata de la relación base, porque esto establece una transformación de cada tupla (virtual) de la vista a una sola tupla base.
- En general, las vistas definidas sobre múltiples tablas por medio de reuniones no son actualizables.
- Las vistas definidas mediante agrupación y funciones agregadas no son actualizables.

En SQL2 se debe agregar la cláusula WITH CHECK OPTION (con opción de verificación) al final de la definición de una vista si se va a actualizar dicha vista. Esto permite al sistema comprobar que la vista sea actualizable y planear una estrategia de ejecución de las actualizaciones.

El problema de implementar con eficiencia una vista para consultarla también es complejo, y se han sugerido dos enfoques. En una estrategia, la **modificación de consultas**, se modifica la consulta de la vista para convertirla en una consulta de las tablas base subyacentes. La desventaja de este enfoque es que es poco eficiente en el caso de vistas definidas mediante consultas complejas cuya ejecución requiere un tiempo apreciable, sobre todo si

se aplican múltiples consultas a la vista dentro de un periodo corto. En la otra estrategia, la **materialización de vistas**, se crea una tabla temporal física de la vista cuando se consulta ésta por primera vez, y se mantiene dicha tabla con la suposición de que se harán más consultas sobre ella. En este caso, es preciso crear una estrategia para actualizar por incrementos la tabla de la vista cuando se modifiquen las tablas base, a fin de mantener la vista al día. Si no se hace referencia a la vista durante cierto tiempo, el sistema podrá eliminar la tabla física de la vista y volverla a calcular cuando consultas futuras se refieran a ella.

7.5 Cómo especificar restricciones adicionales en forma de aserciones*

En SQL2 los usuarios pueden especificar restricciones que no pertenezcan a ninguna de las categorías descritas en la sección 7.1.2 (como las de clave, de integridad de entidades y de integridad referencial) mediante **aserciones declarativas**, empleando la **instrucción** CREATE ASSERTION (crear aserción) del DDL. Cada aserción recibe un nombre de restricción y se especifica con una condición similar a la cláusula WHERE de una consulta SQL. Por ejemplo, para especificar en SQL2 la restricción "El salario de un empleado no debe ser mayor que el salario del gerente del departamento al que pertenece el empleado", podemos escribir la siguiente aserción:

```
CREATE ASSERTION RESTRICC_SALARIO
CHECK ( NOT EXISTS ( SELECT * FROM EMPLEADO E, EMPLEADO G,
                     DEPARTAMENTO D
                     WHERE E.SALARIO>G.SALARIO AND
                     E.ND=D.NÚMEROD AND D.NSSGTE=G.NSS ) );
```

El nombre de restricción RESTRICC_SALARIO va seguido de la palabra reservada **CHECK** (verificar), la cual va seguida de una **condición** entre paréntesis que se debe cumplir en el estado de la base de datos para que se satisfaga la restricción. El nombre de la restricción se puede usar posteriormente para referirse a ella, modificarla o desecharla. El SGBD está obligado a garantizar que no se violará la condición. Se puede usar cualquier condición de cláusula WHERE, pero muchas restricciones se pueden especificar con condiciones estilo EXISTS o NOT EXISTS. Siempre que alguna tupla de la base de datos haga que la condición de una aserción resulte FALSE, la restricción **habrá sido violada**. Un estado de la base de datos **satisface** una restricción si *ninguna combinación de tuplas* de dicho estado viola la restricción. Cabe señalar que la cláusula CHECK y la condición de restricción también pueden usarse junto con la instrucción CREATE DOMAIN (véase la Sec. 7.1.2) a fin de especificar restricciones sobre un dominio específico, como por ejemplo limitar los valores de un dominio a un *subintervalo* del tipo de datos del dominio.

Versiones anteriores de SQL tenían dos tipos de instrucciones para declarar restricciones: ASSERT (aseverar) y TRIGGER (disparar). La **instrucción ASSERT** es parecida a CREATE ASSERTION de SQL2, aunque con una sintaxis distinta. Podemos especificar la restricción "El salario de un empleado no debe ser mayor que el salario del gerente del departamento al que pertenece el empleado" usando ASSERT como sigue:

```
ASSERT RESTRICC_SALARIO ON EMPLEADO E, EMPLEADO G, DEPARTAMENTO D:
NOT (E.SALARIO>G.SALARIO AND E.ND=D.NÚMEROD AND D.NSSGTE=G.NSS);
```


La palabra reservada **ASSERT** indica que se está definiendo una restricción; va seguida del nombre de la restricción, RESTRICC_SALARIO. Después de la palabra reservada ON (en) van los nombres de las relaciones a las que afecta la restricción. Por último, se especifica la condición de la aserción, que también es similar a la condición de una cláusula WHERE.

En muchos casos conviene especificar el tipo de acción que ha de efectuarse en caso de que se viole una restricción. En vez de ofrecer a los usuarios únicamente la opción de abortar la transacción que provoca una violación, el SGBD deberá poner a su disposición otras opciones. Por ejemplo, podría ser útil especificar una restricción que, de violarse, hiciera que se informara de ello al usuario. Es posible que un gerente desee enterarse de los casos en que los viáticos asignados a un empleado excedan un cierto límite, recibiendo un mensaje siempre que esto ocurra. La acción que el SGBD debe emprender en este caso es enviar un mensaje apropiado a ese usuario, con lo que la restricción servirá para **vigilar** la base de datos. Se podrían especificar otras acciones, como ejecutar un determinado procedimiento o disparar otras actualizaciones. Se ha propuesto un mecanismo llamado *disparador* para implementar semejantes acciones en versiones anteriores de SQL.†

Un **disparador** especifica una **condición** y una **acción** que se realizará en caso de que se satisfaga la condición. Ésta casi siempre se especifica en forma de una aserción que invoca o "dispara la acción cuando resulta TRUE. Se ha propuesto una instrucción DEFINE TRIGGER para especificar disparadores. Por ejemplo, podemos especificar un disparador que notifique al gerente de un departamento si algún empleado de éste percibe un salario mayor que el del gerente:

```
DEFINE TRIGGER DISPARADOR_SALARIO ON EMPLEADO E,
EMPLEADO G, DEPARTAMENTO D:
E.SALARIO>G.SALARIO AND E.ND=D.NÚMEROD AND
D.NSSGTE=G.NSS
  ACTION_PROCEDURE INFORMAR_GERENTE (D.NSSGTE);
```

Este disparador especifica que se deberá ejecutar el procedimiento INFORMAR_GERENTE siempre que se cumpla la condición del disparador. Observe la diferencia entre ASSERT y TRIGGER: una instrucción ASSERT prohíbe las actualizaciones que violan la condición de la aserción (la hacen FALSE), en tanto que un disparador permite que se realice la actualización pero ejecuta el procedimiento de acción cuando se cumple la condición del disparador (se vuelve TRUE). Así pues, la condición especificada antes en la RESTRICC_SALARIO mediante ASSERT y la condición de TRIGGER del último ejemplo son mutuamente *inversas*.

Adviértase que los disparadores combinan los enfoques declarativo y por procedimientos con que se implementan las restricciones de integridad. La condición del disparador es declarativa, pero su acción opera por procedimientos.

7.6 Especificación de índices*

Versiones anteriores de SQL tenían instrucciones para crear y desechar índices sobre atributos de las relaciones base. Sin embargo, como los índices son caminos físicos de acceso y no

†En fechas recientes se han desarrollado conceptos para los llamados **sistemas de bases de datos activos** (véase la Sec. 25.3.2), que definen mecanismos generales para incorporar acciones automáticas en el diseño de las bases de datos.

conceptos lógicos, dichas instrucciones se eliminaron de SQL2. En esta sección nos ocupamos de ello porque algunos SGBD todavía cuentan con las instrucciones CREATE INDEX (crear índice). En las primeras versiones, el DDL de SQL no tenía cláusulas para especificar restricciones de clave y de integridad referencial en la instrucción CREATE TABLE. Más bien, la especificación de una *restricción de clave* se combinaba con la especificación de un índice.

Recordemos, del capítulo 5, que un índice es una estructura física de acceso que se especifica con base en uno o más atributos de un archivo. En SQL un archivo corresponde, aproximadamente, a una relación base, de modo que los índices se especifican sobre relaciones base. El atributo o atributos sobre los cuales se crea un índice se denominan **atributos de indización**. Los índices hacen más eficiente el acceso a tuplas con base en condiciones en las que intervienen sus atributos de indización. Esto significa que, en general, la ejecución de una consulta tardará menos si algunos de los atributos implicados en las condiciones de la consulta están indizados. Esta mejoría puede ser impresionante en el caso de consultas de relaciones grandes. En general, si están indizados los atributos que intervienen en las condiciones de selección y de reunión de una consulta, el tiempo de ejecución de ésta se reduce considerablemente.

En los SGBD relacionales, los índices se pueden crear y desechar dinámicamente. La orden **CREATE INDEX** sirve para especificar un índice, el cual recibe un nombre que se usará para desecharlo cuando ya no se necesite. Por ejemplo, para crear un índice sobre el atributo APELLIDO de la relación base EMPLEADO de la figura 6.5, podemos emitir la orden que se muestra en I1:

```
I1:  CREATE  INDEX  ÍNDICE_APELLIDO
             ON     EMPLEADO ( APELLIDO );
```

En general, el índice está en orden ascendente de los valores del atributo de indización. Si queremos los valores en orden descendente, podemos añadir la palabra reservada DESC después del nombre del atributo. La especificación por omisión es ASC, que significa ascendente. También podemos crear un índice sobre una combinación de atributos. Por ejemplo, si queremos crear un índice basado en la combinación de NOMBREP, INIC y APELLIDO, usaremos I2. Aquí suponemos que queremos APELLIDO en orden ascendente y NOMBREP en orden descendente dentro del mismo valor de APELLIDO:

```
I2:  CREATE  INDEX  ÍNDICE_NOMBRES
             ON     EMPLEADO ( APELLIDO ASC, NOMBREP DESC, INIC );
```

SQL ofrece dos opciones adicionales al crear índices. La primera permite especificar la **restricción de clave** sobre el atributo, o combinación de atributos, de indización.† La palabra reservada **UNIQUE** después de la orden CREATE sirve para especificar una clave. Por ejemplo, si queremos especificar un índice sobre el atributo NSS de EMPLEADO y *al mismo tiempo* especificar que NSS es un atributo clave de la relación base EMPLEADO usaremos I3:

```
I3:  CREATE UNIQUE  INDEX  ÍNDICE_NSS
                    ON     EMPLEADO ( NSS );
```

Cabe señalar que lo mejor es especificar las claves antes de insertar tuplas en la relación, para que el sistema pueda hacer cumplir la restricción. Un intento de crear un índice único sobre una tabla base ya existente fracasará si las tuplas que están ya en la tabla no obedecen la restricción de unicidad del atributo de indización.

†Ésta era la única forma de especificar restricciones de clave en las primeras versiones de SQL.

La razón para vincular la definición de una restricción de clave con la especificación de un índice en las primeras versiones de SQL era que es mucho más eficiente imponer a un archivo la unicidad de los valores de las claves si se ha definido un índice sobre el atributo clave. Podemos comprobar si existen valores repetidos en el índice al realizar búsquedas en él. Si no existe índice, en la mayoría de los casos será necesario examinar todo el archivo para averiguar si un atributo tiene algún valor repetido.

La segunda opción al crear un índice permite especificar si se trata o no de un índice de agrupamiento (véase el Cap. 5). Las condiciones de reunión y de selección son todavía más eficientes cuando se especifican en términos de un atributo que tiene un índice de agrupamiento. En este caso se usa la palabra reservada **CLUSTER** (grupo) al final de la orden CREATE INDEX. Por ejemplo, si queremos que los registros EMPLEADO estén indizados y agrupados por número de departamento, crearemos un índice de agrupamiento sobre ND, como en I4:

```
I4:  CREATE  INDEX  ÍNDICE_ND
             ON     EMPLEADO ( ND )
             CLUSTER;
```

Una relación base puede tener *como máximo un* índice de agrupamiento, pero cualquier número de índices que no sean de agrupamiento.

Un *índice de agrupamiento y único* es similar al *índice primario* del capítulo 5. Un *índice de agrupamiento no único* es similar al *índice de agrupamiento* del capítulo 5. Por último, un *índice que no es de agrupamiento* es similar al *índice secundario* del capítulo 5. Cada SGBD puede tener su propio tipo de técnica de implementación de índices; por ejemplo, los índices multiniveles pueden usar árboles B o árboles B^+, o algunas otras variaciones similares a las que vimos en el capítulo 5. Muchos SGBD relacionales ahora cuentan con estructuras de almacenamiento basadas en dispersión.

La orden **DROP INDEX** sirve para desechar un índice. Los índices se desechan porque su mantenimiento resulta costoso cada vez que se actualiza la relación base y requieren almacenamiento adicional. Por ello, si ya no esperamos emitir consultas en las que intervenga un atributo indizado, nos conviene desechar ese índice. He aquí un ejemplo de eliminación del índice ND:

```
I5:  DROP INDEX  ÍNDICE_ND;
```

7.7 SQL incorporado*

SQL también puede usarse junto con un lenguaje de programación de aplicación general como C, ADA, PASCAL, COBOL o PL/I. El lenguaje de programación se denomina **lenguaje anfitrión**. Cualquier instrucción de SQL —de definición de datos, consulta, actualización o definición de vistas— se puede incorporar en un programa escrito en el lenguaje anfitrión. La instrucción de SQL incorporado se distingue de las instrucciones del lenguaje anfitrión anteponiéndole un carácter o una orden especial como prefijo para que el **preprocesador** pueda separar las instrucciones de SQL incorporadas y el código del lenguaje anfitrión. En SQL2, las palabras reservadas EXEC SQL o la secuencia &SQL(preceden a todo enunciado de SQL incorporado, y las instrucciones pueden terminar con un END-EXEC. correspondiente, un ")" o un ";". En algunas de las primeras implementaciones, las instrucciones de SQL se pasaban como parámetros en llamadas a procedimientos.

†En algunas de las primeras versiones de SQL se usaba el signo "%".

En general, los diferentes sistemas siguen distintas convenciones para incorporar instrucciones de SQL. A fin de ilustrar los conceptos del SQL incorporado, adoptaremos como lenguaje anfitrión a PASCAL y definiremos nuestra propia sintaxis. Usaremos un signo "$" para identificar las instrucciones de SQL en el programa y ";" será el carácter terminador. Dentro de una orden de SQL incorporado, podemos hacer referencia a variables del programa, a las que antepondremos un signo ":".† Esto permitirá que las variables del programa y los objetos del esquema de base de datos (los atributos y las relaciones) tengan los mismos nombres sin que haya ambigüedad.

Suponga que deseamos escribir programas en PASCAL para procesar la base de datos que aparece en la figura 6.5. Necesitaremos declarar variables de programa con los mismos tipos que los atributos de la base de datos que el programa va a procesar. Estas variables pueden o no tener nombres que sean idénticos a los de sus atributos correspondientes. Usaremos las variables de programa PASCAL declaradas en la figura 7.6 para todos nuestros ejemplos, y mostraremos los segmentos de programas en PASCAL sin declaraciones de variables.

Como primer ejemplo, escribiremos un segmento de programa repetitivo (ciclo) para leer un número de seguro social e imprimir cierta información de la tupla EMPLEADO correspondiente. El código de programa en PASCAL se muestra en E1, en donde suponemos que en otro lugar se han declarado las variables de programa apropiadas NÚM_SEG_SOC y CICLO. El programa lee un número de seguro social y a continuación obtiene la tupla EMPLEADO que tiene ese número de seguro social mediante la orden SQL incorporada. Las órdenes de obtención de datos de SQL incorporado requieren una cláusula INTO (en) para especificar las variables de programa en las que se colocarán los valores de atributos obtenidos de la base de datos. Las variables de programa en PASCAL de la cláusula INTO llevan el prefijo ":". He aquí el segmento de programa E1:

```
E1:  CICLO:= 'S';
     while CICLO = 'S' do
        begin
        writeln('introduzca un número de seguro social:');
        readln(NÚM_SEG_SOC);
        $SELECT NOMBREP, INIC, APELLIDO, DIRECCIÓN, SALARIO
        INTO :E.NOMBREP, :E.INIC, :E.APELLIDO, :E.DIRECCIÓN, :E.SALARIO
        FROM EMPLEADO
        WHERE NSS=:NÚM_SEG_SOC;
        writeln(E.NOMBREP, E.INIC, E.APELLIDO, E.DIRECCIÓN, E.SALARIO);
        writeln('¿más números de seguro social (S o N)? ');
        readln(CICLO)
        end;
```

En E1 la consulta de SQL incorporada selecciona una sola tupla; por eso, podemos asignar directamente los valores de sus atributos a variables del programa. En general, una consulta

```
var NOMBRED: packed array[1..15] of char;
    NÚMEROD, AUMENTO: integer;
    E: record NOMBREP, APELLIDO: packed array[1..15] of char;
              INIC, SEXO: char;
              NSS, FECHAN, NSSSUPER: packed array[1..9] of char;
              DIRECCIÓN: packed array[1..30] of char;
              SALARIO, ND: integer
       end;
```

Figura 7.6 Variables de programa PASCAL utilizadas en E1 y E2.

de SQL puede obtener muchas tuplas, en cuyo caso el programa en PASCAL por lo regular examinará todas las tuplas obtenidas y las procesará una por una. Se utiliza el concepto de cursor para que el programa escrito en el lenguaje anfitrión pueda procesar las tuplas una por una.

Podemos concebir un **cursor** como un apuntador que apunta a *una sola tupla (fila)* del resultado de una consulta. El cursor se declara cuando se declara la orden de consulta de SQL en el programa. Más adelante, una orden OPEN (abrir) cursor obtiene de la base de datos el resultado de la consulta y ajusta el cursor a una posición *antes de la primera fila* del resultado de la consulta. Ésta se convierte en la **fila actual** para el cursor. Posteriormente, se emiten órdenes FETCH (traer) en el programa, y cada una avanza el cursor a la *siguiente fila* del resultado de la consulta, convirtiéndola en la fila actual y copiando los valores de sus atributos en las variables de programa de PASCAL especificadas en la orden FETCH. Esto es similar al procesamiento de archivos tradicional de registro por registro.

Para determinar si ya se procesaron todas las tuplas del resultado de la consulta se utiliza una variable implícita, llamada SQLCODE,† para comunicar al programa la situación de las órdenes de SQL incorporadas. Cuando se ejecuta con éxito una orden de SQL, se devuelve un código de 0 (cero) en SQLCODE; se devuelven códigos diferentes para indicar excepciones y errores. Si se emite una orden FETCH que desplaza el cursor más allá de la última tupla del resultado de la consulta, se devuelve un código especial END_OF_CURSOR. El programador utiliza esto para terminar un ciclo de procesamiento con las tuplas del resultado de una consulta. En general, es posible tener abiertos muchos cursores al mismo tiempo. Para indicar que ya no vamos a utilizar el resultado de una consulta se emite una orden CLOSE (cerrar) cursor.

En E2 se muestra un ejemplo del empleo de cursores, y en él se declara explícitamente el cursor EMP. Suponemos que ya se declararon las variables de programa de PASCAL apropiadas (Fig. 7.6). El segmento de programa E2 lee un nombre de departamento y exhibe los nombres de los empleados que trabajan en ese departamento, uno por uno. El programa lee un importe de aumento para el salario de cada empleado y actualiza este último en esa cantidad:

```
E2:  writeln('introduzca el nombre del departamento:'); readln (NOMBRED);
     $SELECT NÚMEROD INTO :NÚMEROD
     FROM DEPARTAMENTO
     WHERE NOMBRED=:NOMBRED;
     $DECLARE    EMP CURSOR FOR
                 SELECT NSS, NOMBREP, INIC, APELLIDO, SALARIO
                 FROM EMPLEADO
                 WHERE ND=:NÚMEROD
     FOR UPDATE OF SALARIO;
     $OPEN EMP;
     $FETCH EMP INTO :E.NSS, :E.NOMBREP, :E.INIC, :E.APELLIDO,
     :E.SALARIO;
     while SQLCODE = 0 do
        begin
        writeln('nombre del empleado: ', E.NOMBREP, E.INIC, E.APELLIDO);
        writeln('introduzca aumento: '); readln(AUMENTO);
        $UPDATE EMPLEADO SET SALARIO = SALARIO + :AUMENTO
                        WHERE CURRENT OF EMP;
        $FETCH EMP INTO :E.NSS, :E.NOMBREP, :E.INIC, :E.APELLIDO,
```

†Ésta se llama ahora SQLSTATE en SQL2.

```
        :E.SALARIO;
        end;
    $CLOSE CURSOR EMP;
```

En SQL2, además de tener acceso puramente secuencial, es posible colocar el cursor de otras maneras. Se puede agregar una **orientación de obtención** cuyos valores pueden ser NEXT (siguiente), PRIOR (anterior), FIRST (primero), LAST (último), ABSOLUTE *i* (*i* absoluto) y RELATIVE i (*i* relativo). En las últimas dos órdenes, el resultado de evaluar *i* debe ser un valor entero que especifique una posición absoluta de tupla o una posición de tupla relativa a la posición actual del cursor. La orientación de obtención por omisión es NEXT. Esto permite al programador desplazar el cursor entre las tuplas del resultado de la consulta con mayor flexibilidad, realizando acceso aleatorio por posición. La forma general de la orden FETCH es la siguiente (las partes entre corchetes son opcionales):

FETCH [[<orientación de obtención>] **FROM**]
<nombre de cursor> **INTO** <lista destino de obtención>;

Cuando se define un cursor para filas que se van a modificar, hay que añadir la cláusula **FOR UPDATE OF** (para modificación de) en la declaración del cursor y listar los nombres de todos los atributos que el programa vaya a modificar. Esto se ilustra en E2. Si se van a eliminar filas, hay que añadir las palabras FOR DELETE. En la orden UPDATE (o DELETE) incorporada, la condición **WHERE CURRENT OF** (donde actual de) cursor especifica que la tupla actual es la que se modificará (o eliminará).

7.8 Resumen

En este capítulo presentamos el lenguaje de consulta de bases de datos SQL. Este lenguaje o sus variaciones se han implementado como interfaces de varios SGBD relacionales que se encuentran en el mercado, entre ellos DB2 y SQL/DS de IBM, ORACLE, INGRES y UNIFY. La versión original de SQL se implementó en el SGBD experimental llamado SYSTEM R, desarrollado en IBM Research. SQL está diseñado como un lenguaje amplio que incluye instrucciones para definición de datos, consultas, actualizaciones y definición de vistas. Estudiaremos cada uno de estos temas en secciones individuales del capítulo. Nos basamos principalmente en la norma SQL2, pero también vimos algunas características de versiones menos recientes del lenguaje, como CREATE INDEX, a fin de ofrecer una perspectiva histórica apropiada.

En la última sección tratamos la incorporación de SQL en un lenguaje de programación de aplicación general. Presentamos el concepto de cursor, con el que un programador puede procesar el resultado de una consulta de alto nivel tupla por tupla.

La tabla 7.1 muestra un resumen de la sintaxis o la estructura de varias instrucciones de SQL. No pretendemos que esta sinopsis sea exhaustiva, ni que describa todas las construcciones posibles en SQL; más bien, esperamos que sirva como referencia rápida sobre los principales tipos de construcciones disponibles en el lenguaje SQL. Adoptamos la notación BNF, en la que los símbolos no terminales aparecen entre paréntesis angulares <. . .>, las partes opcionales se encierran en corchetes [. . .], las repeticiones aparecen entre claves {. . .} y las alternativas se incluyen entre paréntesis (. . .|. . .|. . .).†

†La sintaxis completa de SQL2 se describe en un documento de más de 500 páginas.

Tabla 7.1 Resumen de la sintaxis de SQL

```
CREATE TABLE <nombre de tabla> (<nombre de columna> <tipo de columna>
                                [<restricción de atributo>]
                                {, <nombre de columna> <tipo de columna>
                                [<restricción de atributo>]}
                                [<restricción de tabla> {,<restricción de tabla>}])

DROP TABLE <nombre de tabla>

ALTER TABLE <nombre de tabla> ADD <nombre de columna> <tipo de columna>

SELECT [DISTINCT] <lista de atributos>
FROM (<nombre de tabla> {<seudónimo>}|<tabla reunida>) {, (<nombre de tabla>
{<seudónimo>}|<tabla reunida>)}
[WHERE <condición>]
[GROUP BY <atributos de agrupación>[HAVING <condición de selección de grupo>]]
[ORDER BY <nombre de columna> [<orden>] {, <nombre de columna> [<orden>]}]

<lista de atributos>::=(*|(<nombre de columna>|<función>(([DISTINCT]<nombre de
columna>|*)))
                        {, (<nombre de columna>|<función>(([DISTINCT] <nombre de
                        columna>|*))))
<atributos de agrupación>::=<nombre de columna> {, <nombre de columna>}
<orden>::=(ASC|DESC)

INSERT INTO <nombre de tabla> [(<nombre de columna> {, <nombre de columna>})]
(VALUES(<valor constante>, {<valor constante>}){,(<valor constante>
{,<valor constante>})}

|<instrucción de selección>)

DELETE FROM <nombre de tabla>
[WHERE <condición de selección>]

UPDATE <nombre de tabla>
SET <nombre de columna>=<expresión de valor> {, <nombre de columna>=<expresión
de valor>}
[WHERE <condición de selección>]

CREATE [UNIQUE] INDEX <nombre de índice>
ON <nombre de tabla>(<nombre de columna>[<orden>]{, <nombre de columna>
[<orden>]})
[CLUSTER]

DROP INDEX <nombre de índice>

CREATE VIEW <nombre de vista>[(<nombre de columna> {, <nombre de columna>})]
AS <instrucción de selección>

DROP VIEW <nombre de vista>
```

Preguntas de repaso

7.1. ¿Qué diferencia hay entre las relaciones (tablas) de SQL y las relaciones que definimos formalmente en el capítulo 6? Explique las demás diferencias en terminología.

7.2. ¿Por qué SQL permite tuplas repetidas en una tabla o en el resultado de una consulta?

7.3. ¿Por qué las primeras implementaciones de SQL sólo permitían definir una clave junto con un índice?

7.4. ¿Cómo permite SQL la implementación de las restricciones de integridad de entidades e integridad referencial que describimos en el capítulo 6? ¿Qué sucede con las restricciones de integridad generales?

7.5. ¿Qué es una vista en SQL y cómo se define? Analice los problemas que pueden surgir cuando se intenta actualizar una vista. ¿Cómo se implementan las vistas?

7.6. ¿Qué es un cursor? ¿Cómo se usa en SQL incorporado? ¿Cómo se colocan los cursores para tener acceso no secuencial a las filas?

Ejercicios

7.7. Considere la base de datos que aparece en la figura 1.2, cuyo esquema se muestra en la figura 2.1. ¿Qué restricciones de integridad referencial deben cumplirse en este esquema? Escriba instrucciones apropiadas en el DDL de SQL para definir la base de datos.

7.8. Repita el ejercicio 7.7, pero utilice el esquema de base de datos AEROLÍNEA de la figura 6.20.

7.9. Considere el esquema de base de datos relacional BIBLIOTECA de la figura 6.22. Escoja la acción apropiada (rechazar, propagar, poner nulo, poner valor por omisión) para cada restricción de integridad referencial, tanto para la *eliminación* de una tupla referida como para la *modificación* de un valor de atributo de clave primaria en la tupla referida. Justifique sus elecciones.

7.10. Escriba instrucciones apropiadas en el DDL de SQL para declarar el esquema de la base de datos relacional BIBLIOTECA de la figura 6.22. Utilice la acción referencial elegida en el ejercicio 7.9.

7.11. Escriba consultas SQL para las consultas de la base de datos BIBLIOTECA dadas en el ejercicio 6.26.

7.12. ¿Cómo puede el SGBD imponer las restricciones de clave y de clave externa? ¿Es difícil implementar la técnica de imposición que usted sugirió? ¿Es posible efectuar de manera eficiente las comprobaciones de restricción cuando se aplican actualizaciones a la base de datos?

7.13. Especifique las consultas del ejercicio 6.19 en SQL. Muestre el resultado de cada una de las consultas si se aplica a la base de datos COMPAÑÍA de la figura 6.6.

7.14. Especifique en SQL las siguientes consultas adicionales a la base de datos de la figura 6.5. Muestre los resultados de cada consulta si ésta se aplica a la base de datos de la figura 6.6.

a. Para cada departamento en el que el salario medio de los empleados sea mayor que $30 000, obtener el nombre del departamento y el número de empleados que pertenecen a él.

b. Suponga que desea conocer el número de empleados *de sexo masculino* en cada departamento, en vez de todos los empleados (como en el ejercicio 7.14a). ¿Es posible especificar esta consulta en SQL? ¿Por qué sí o por qué no?

7.15. Especifique las actualizaciones del ejercicio 6.20, empleando las órdenes de actualización de SQL.

7.16. Especifique en SQL las siguientes consultas relativas al esquema de base de datos que se ve en la figura 2.1.

a. Obtener los nombres de todos los estudiantes de cuarto año de la carrera 'CICO' (ciencias de la computación).

b. Obtener los nombres de todos los cursos impartidos por el profesor Reyes en 1985 y 1986.

c. Para cada sección impartida por el profesor Reyes, obtener el número del curso, el semestre, el año y el número de estudiantes que tomaron la sección.

d. Obtener el nombre y la boleta de notas de todos los estudiantes de último año (Grado = 5) de la carrera CICO. La boleta incluye el nombre del curso, el número del curso, las horas-crédito, el semestre, el año y las notas de cada uno de los cursos concluidos por el estudiante.

e. Obtener los nombres y departamentos de carrera de todos los estudiantes que hayan obtenido la nota A en todos sus cursos.

f. Obtener los nombres y departamentos de carrera de todos los estudiantes que no tengan una nota A en ninguno de sus cursos.

7.17. Escriba instrucciones de actualización de SQL para realizar las siguientes operaciones sobre el esquema de base de datos de la figura 2.1.

a. Insertar un nuevo estudiante <'Jiménez', 25, 1, 'MATE'> en la base de datos.

b. Cambiar el grado del estudiante 'Silva' a 2.

c. Insertar un nuevo curso <'Ingeniería del conocimiento', 'CICO4390', 3, 'CICO'>.

d. Eliminar el registro del estudiante de nombre 'Silva' con número de estudiante 17.

7.18. Escriba programas en PASCAL con instrucciones de SQL incorporado, siguiendo el estilo que se mostró en la sección 7.7, para realizar las siguientes tareas sobre el esquema de base de datos de la figura 2.1. Defina variables de programa apropiadas para cada codificación.

a. Introducir las notas de los estudiantes de una sección. El programa deberá capturar el identificador de sección y luego iniciar un ciclo que capture el número de cada estudiante y su nota; insertar esta información en la base de datos.

b. Imprimir la boleta de notas de un estudiante. El programa deberá capturar el identificador del estudiante e imprimir su nombre y una lista de <número de curso, nombre de curso, identificador de sección, semestre, año, nota> por cada sección que haya concluido dicho estudiante.

7.19. Escriba instrucciones para crear índices sobre el esquema de base de datos de la figura 2.1, según los siguientes atributos:

a. Un índice de agrupamiento único sobre el atributo NúmEstudiante de ESTUDIANTE.

b. Un índice de agrupamiento sobre el atributo NúmEstudiante de INFORME_NOTAS.

c. Un índice sobre el atributo Carrera de ESTUDIANTE.

7.20. ¿Qué tipos de consultas se volverían más eficientes con cada uno de los índices especificados en el ejercicio 7.19?

7.21. Especifique las siguientes vistas en SQL sobre el esquema de base de datos COMPAÑÍA que se muestra en la figura 6.5.

a. Una vista con el nombre de departamento, nombre del gerente y salario del gerente para cada departamento.

b. Una vista con el nombre de empleado, el nombre del supervisor y el salario de cada empleado que trabaja en el departamento 'Investigación'.

c. Una vista con el nombre de proyecto, el nombre del departamento controlador, el número de empleados y el total de horas trabajadas por semana en el proyecto para cada proyecto.

d. Una vista con el nombre de proyecto, el nombre del departamento controlador, el número de empleados y el total de horas trabajadas por semana en el proyecto para cada proyecto *en el que trabajan dos o más empleados*.

7.22. Considere la vista RESUMEN_DEPTO, definida sobre la base de datos COMPAÑÍA de la figura 6.6:

```
CREATE VIEW    RESUMEN_DEPTO (D, C, S_TOTAL, S_MEDIO)
AS SELECT      ND, COUNT (*), SUM (SALARIO), AVG (SALARIO)
   FROM        EMPLEADO
   GROUP BY    ND;
```

Indique cuáles de las siguientes consultas y actualizaciones se permitirían en la vista. Si una consulta o actualización está permitida, indique el aspecto que tendría la consulta o actualización correspondiente en las relaciones base, y muestre el resultado de aplicarla a la base de datos de la figura 6.6.

a.
```
SELECT    *
FROM      RESUMEN_DEPTO
```

b.
```
SELECT    D, C
FROM      RESUMEN_DEPTO
WHERE     S_TOTAL > 100000
```

c.
```
SELECT    D, S_MEDIO
FROM      RESUMEN_DEPTO
WHERE     C > (SELECT C FROM RESUMEN_DEPTO WHERE D=4)
```

d.
```
UPDATE    RESUMEN_DEPTO
SET       D=3
WHERE     D=4
```

e.
```
DELETE FROM  RESUMEN_DEPTO
WHERE     C>4
```


7.23. Considere el esquema de relación CONTIENE(Núm_comp_padre, Núm_sub_comp); una tupla $<C_i, C_j>$ en CONTIENE significa que el componente C_i contiene al componente C_j como componente directo. Suponga que escoge un componente C_k que no contiene otros componentes y quiere averiguar los números de componente de todos los componentes que contienen a C_k, directa o indirectamente, en cualquier nivel. Ésta es una *consulta recursiva* que requiere el cálculo de la **cerradura transitiva** de CONTIENE. Demuestre que esta consulta no se puede especificar directamente como una sola consulta SQL. ¿Puede sugerir extensiones del lenguaje que permitan la especificación de tales consultas?

7.24. Especifique en SQL las consultas y actualizaciones de los ejercicios 6.21 y 6.22, que se refieren a la base de datos AEROLÍNEA.

7.25. Escoja alguna aplicación de base de datos que conozca bien.

a. Diseñe un esquema de base de datos relacional para su aplicación.

b. Declare sus relaciones, empleando el DDL de SQL.

c. Especifique varias consultas que necesite su aplicación de base de datos en SQL.

d. Con base en el uso que se piensa dar a la base de datos, escoja algunos atributos sobre los cuales convendría especificar índices.

e. Implemente su base de datos, si usted dispone de un sistema SQL.

Bibliografía selecta

El lenguaje SQL, originalmente llamado SEQUEL, fue un sucesor del lenguaje SQUARE (*Specifying Queries as Relational Expressions:* especificación de consultas como expresiones relacionales), descrito por Boyce *et al.* (1975). La sintaxis de SQUARE se modificó para crear SEQUEL (Chamberlin y Boyce, 1974) y luego SEQUEL 2 (Chamberlin *et al.*, 1976), en los que se basó SQL. La implementación original de SEQUEL se realizó en IBM Research, San Jose, California.

Reisner (1977) describe una evaluación sobre los factores humanos que influyeron en SEQUEL; descubrió que para los usuarios es problemático especificar correctamente condiciones de reunión y agrupación. Date (1984*b*) contiene una crítica del lenguaje SQL que destaca sus ventajas y deficiencias. Date y Darwen (1993) describe SQL2. ANSI (1986) bosqueja la norma SQL original, y ANSI (1992) explica la nueva norma SQL2. Diversos manuales de proveedores describen las características de SQL tal como se implementó en DB2, SQL/DS, ORACLE, INGRES, UNIFY y otros productos de SGBD comerciales. Horowitz (1992) estudia algunos de los problemas relacionados con la integridad referencial y la propagación de actualizaciones en SQL2.

El problema de las actualizaciones de vistas se aborda en Dayal y Bernstein (1978), Keller (1982) y Langerak (1990), entre otros. La implementación de vistas se analiza en Blakeley *et al.* (1989) y Roussopoulos (1990). Negri *et al.* (1991) describe la semántica formal de las consultas SQL.

CAPÍTULO 8

Cálculo relacional, QUEL y QBE

En el capítulo 6 presentamos el modelo relacional de los datos y examinamos las operaciones del álgebra relacional, fundamentales para manipular una base de datos relacional. Aunque describimos el álgebra relacional como parte integral de dicho modelo, en realidad no es más que un tipo de lenguaje de consulta formal para especificar obtenciones de datos y formar nuevas relaciones a partir de una base de datos relacional. En este capítulo estudiaremos otro lenguaje formal para las bases de datos relacionales, el **cálculo relacional**. En el mercado hay muchos lenguajes de bases de datos relacionales que se apoyan en algunos aspectos de este cálculo, entre ellos el lenguaje SQL que vimos en el capítulo 7. Sin embargo, algunos lenguajes son más parecidos al cálculo relacional que otros; por ejemplo, los lenguajes QUEL y QBE que presentaremos en las secciones 8.2 y 8.4 están más cerca del cálculo relacional que SQL.

¿En qué radica la diferencia entre el cálculo relacional y el álgebra relacional? La principal diferencia es que en el cálculo escribimos una expresión **declarativa** para especificar una solicitud de obtención de datos, en tanto que en el álgebra relacional debemos escribir una *secuencia de operaciones*. Es cierto que estas operaciones se pueden anidar para formar una sola expresión, pero siempre se especifica explícitamente un cierto orden de las operaciones en una expresión del álgebra relacional. Este orden también especifica una estrategia parcial para evaluar la consulta. En el cálculo relacional no se describe cómo evaluar una consulta; una expresión del cálculo especifica *qué* debe obtenerse, no *cómo* debe hacerse. Por tanto, el cálculo relacional se considera un lenguaje declarativo, es decir, que **no funciona por procedimientos**.

En un sentido importante, el cálculo y el álgebra relacionales son idénticos. Se ha demostrado que cualquier obtención de datos que se pueda especificar en el álgebra relacional puede especificarse también en el cálculo relacional, y viceversa; en otras palabras, el **poder de expresión** de los dos lenguajes es *idéntico*. Esto ha dado lugar a la definición del concepto de lenguaje relacionalmente completo. Se dice que un lenguaje de consulta relacional L es **relacionalmente completo** si es posible expresar en L cualquier consulta que se pueda

expresar en el cálculo relacional. La compleción relacional se ha convertido en un parámetro decisivo para comparar el poder de expresión de los lenguajes de consulta de alto nivel. No obstante, como vimos en la sección 6.6, hay algunas consultas que se requieren con frecuencia en las aplicaciones de bases de datos que no se pueden expresar ni en el álgebra ni en el cálculo relacionales. La mayoría de los lenguajes de consulta relacionales son relacionalmente completos, pero tienen *mayor poder de expresión* que el álgebra o el cálculo relacionales gracias a operaciones adicionales como las funciones agregadas, la agrupación y la ordenación.

El cálculo relacional es un lenguaje formal, basado en la rama de la lógica matemática llamada cálculo de predicados. Hay dos formas bien conocidas de adaptar el cálculo de predicados para crear un lenguaje para las bases de datos relacionales. La primera se denomina **cálculo relacional de tuplas**, y la segunda, **cálculo relacional de dominios**. Ambas son adaptaciones del cálculo de predicados de primer orden.[†] En el cálculo relacional de tuplas, las variables abarcan tuplas, mientras que en el cálculo relacional de dominios las variables abarcan valores del dominio de los atributos. Esto lo veremos en las secciones 8.1 y 8.3, respectivamente. Una vez más, todos nuestros ejemplos se referirán a la base de datos de las figuras 6.5 y 6.6.

En este capítulo también presentaremos dos de los primeros lenguajes de consulta relacionales que se han destacado por razones históricas. El lenguaje de consulta QUEL se creó originalmente al mismo tiempo que SQL, pero ha perdido terreno, pese a su popularidad entre los investigadores, debido a la necesidad de estandarizarlo. La importancia del lenguaje QBE se debe a que es uno de los primeros lenguajes de consulta gráficos creados para sistemas de bases de datos. Presentaremos un resumen de los conceptos de QUEL en la sección 8.2, y en la sección 8.4, un panorama de QBE.

El lector puede omitir las secciones 8.2, 8.3 y 8.4 si lo único que desea es una breve introducción al cálculo relacional.

8.1 Cálculo relacional de tuplas

8.1.1 *Variables de tupla y relaciones de intervalo*

El cálculo relacional de tuplas se basa en la especificación de un cierto número de **variables de tupla**. Cada variable de tupla por lo regular **abarca** una determinada relación de una base de datos, lo que significa que la variable puede adoptar como valor cualquier tupla individual de esa relación. Una consulta simple del cálculo relacional tiene la forma

$\{t \mid \text{COND}(t)\}$

donde t es una variable de tupla y COND(t) es una expresión condicional en la que interviene t. El resultado de una consulta como ésta es el conjunto de todas las tuplas t que satisfacen COND(t). Por ejemplo, si queremos averiguar cuáles son los empleados cuyo salario rebasa los \$50 000, podemos escribir la siguiente expresión del cálculo relacional de tuplas:

$\{t \mid \text{EMPLEADO}(t)$ **and** $t.\text{SALARIO}>50000\}$

[†]En nuestra exposición, *no* supondremos que el lector conoce el cálculo de predicados de primer orden, que se ocupa de variables y valores cuantificados. Si queremos manejar relaciones cuantificadas o conjuntos de conjuntos, tendremos que utilizar un cálculo de predicados de orden superior.

La condición EMPLEADO(t) especifica que la *relación de intervalo* de (t *relación abarcada* por) la variable de tupla t es EMPLEADO. Se obtendrá toda tupla t de EMPLEADO que satisfaga la condición t.SALARIO>50000. Observe que t.SALARIO hace referencia al atributo SALARIO de la variable de tupla t. Esta notación se asemeja a la forma en que los nombres de atributos se califican con nombres de relaciones o seudónimos en SQL. En la notación del capítulo 6, t.SALARIO equivale a escribir t[SALARIO].

La consulta anterior obtiene los valores de todos los atributos de cada una de las tuplas EMPLEADO t seleccionadas. Si queremos obtener sólo *algunos* de los atributos —digamos el nombre de pila y el apellido— escribiremos

{t.NOMBREP, t.APELLIDO | EMPLEADO(t) **and** t.SALARIO>50000}

Esto equivale a la siguiente consulta en SQL:

```
SELECT  T.NOMBREP, T.APELLIDO
FROM    EMPLEADO T
WHERE   T.SALARIO>50000
```

En términos informales, necesitamos especificar la siguiente información en una expresión del cálculo relacional de tuplas:

1. Para cada variable de tupla t, la **relación de intervalo** R de t. Este valor se especifica mediante una condición de la forma $R(t)$.
2. Una condición para seleccionar combinaciones específicas de tuplas. Conforme las variables de tupla abarcan sus relaciones de intervalo respectivas, la condición se evalúa para cada posible combinación de tuplas a fin de identificar las **combinaciones seleccionadas** para las cuales la evaluación de la condición resulta TRUE.
3. Un conjunto de atributos que se obtendrán, los **atributos solicitados**. Se obtienen los valores de estos atributos para cada combinación de tuplas seleccionada.

Observe la correspondencia de los elementos anteriores con una consulta SQL simple: el elemento 1 corresponde a los nombres de relación de la cláusula FROM; el elemento 2 corresponde a la condición de la cláusula WHERE, y el elemento 3 corresponde a la lista de atributos de la cláusula SELECT. Antes de examinar la sintaxis formal del cálculo relacional de tuplas, consideremos otra consulta que ya hemos visto.

CONSULTA 0
Obtener la fecha de nacimiento y la dirección del empleado (o empleados) cuyo nombre es 'José B. Silva'.

C0: {t.FECHAN, t.DIRECCIÓN | EMPLEADO(t) **and** t.NOMBREP='José' **and** t.INIC='B' **and** t.APELLIDO='Silva'}

En el cálculo relacional de tuplas, primero especificamos los atributos solicitados t.FECHAN y t.DIRECCIÓN por cada tupla seleccionada t. Luego, después de la barra (|), especificamos la condición para seleccionar una tupla, a saber, que t sea una tupla de la relación EMPLEADO cuyos valores de los atributos NOMBREP, INIC y APELLIDO sean 'José', 'B' y 'Silva', respectivamente.

8.1.2 *Especificación formal del cálculo relacional de tuplas*

Una **expresión** general del cálculo relacional de tuplas tiene la forma

$$\{t_1.A_1, t_2.A_2, \ldots, t_n.A_n \mid \text{COND}(t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m})\}$$

donde $t_1, t_2, \ldots, t_n, t_{n+1}, t_{n+2}, \ldots, t_{n+m}$ son variables de tupla, cada A_i es un atributo de la relación que abarca t, y COND es una **condición** o **fórmula**[†] del cálculo relacional de tuplas. Una fórmula está constituida por **átomos** del cálculo de predicados, que pueden ser uno de los siguientes:

1. Un átomo de la forma $R(t_i)$, donde R es un nombre de relación y t_i es una variable de tupla. Este átomo identifica el intervalo de la variable de tupla t_i como la relación cuyo nombre es R.
2. Un átomo de la forma $t_i.A$ **op** $t_j.B$, donde **op** es uno de los operadores de comparación en el conjunto $\{=, \neq, <, \leq, >, \geq\}$, t_i y t_j son variables de tupla, A es un atributo de la relación que t_i abarca, y B es un atributo de la relación que t_j abarca.
3. Un átomo de la forma $t_i.A$ **op** c o c **op** $t_j.B$, donde **op** es uno de los operadores de comparación del conjunto $\{=, \neq, <, \leq, >, \geq\}$, t_i y t_j son variables de tupla, A es un atributo de la relación que t_i abarca, B es un atributo de la relación que t_j abarca y c es un valor constante.

Si evaluamos cualquiera de los átomos anteriores para una combinación específica de tuplas, obtendremos TRUE o bien FALSE; esto se denomina **valor lógico** de un átomo. En general, una variable de tupla abarca todas las posibles tuplas "en el universo". En el caso de átomos del tipo 1, si asignamos a la variable de tupla una tupla que es *miembro de la relación especificada* R, el átomo será TRUE; de lo contrario será FALSE. En el caso de átomos de los tipos 2 y 3, si las variables de tupla se asignan a tuplas tales que los valores de los atributos especificados de las tuplas satisfagan la condición, el átomo será TRUE.

Una **fórmula** (condición) consta de uno o más átomos conectados mediante los operadores lógicos **and** (y), **or** (o) y **not** (no) y se define recursivamente así:

1. Todo átomo es una fórmula.
2. Si F_1 y F_2 son fórmulas, también son fórmulas (F_1 **and** F_2), (F_1 **or** F_2), **not**(F_1) y **not**(F_2). Los valores lógicos de estas cuatro fórmulas se derivan de sus fórmulas componentes F_1 y F_2 como sigue:
   a. (F_1 **and** F_2) es TRUE si tanto F_1 como F_2 son TRUE; de lo contrario, es FALSE.
   b. (F_1 **or** F_2) es FALSE si tanto F_1 como F_2 son FALSE; de lo contrario, es TRUE.
   c. **not**(F_1) es TRUE si F_1 es FALSE; es FALSE si F_1 es TRUE.
   d. **not**(F_2) es TRUE si F_2 es FALSE; es FALSE si F_2 es TRUE.

Además, hay dos símbolos especiales, llamados **cuantificadores**, que pueden aparecer en las fórmulas; éstos son el **cuantificador universal** ($\forall$) y el **cuantificador existencial** ($\exists$). Los valores lógicos de las fórmulas con cuantificadores se describen en los incisos 3 y 4 (veáse

[†]También se le conoce como **fórmula bien formada**, o **fbf**, en lógica matemática.

más adelante), pero antes necesitamos definir los conceptos de variable de tupla libre y variable de tupla ligada en una fórmula. En términos informales, una variable de tupla t está ligada si está cuantificada; esto es, si aparece en una cláusula $(\exists\, t)$ o $(\forall\, t)$; de lo contrario, está libre. En términos formales, definimos una variable de tupla en una fórmula como **libre** o **ligada** de acuerdo con las siguientes reglas:

- Una ocurrencia de una variable de tupla en una fórmula F que sea *un átomo* está libre en F.
- Una ocurrencia de una variable de tupla t está libre o ligada en una fórmula constituida por conectores lógicos —(F_1 **and** F_2), (F_1 **or** F_2), **not**(F_1) y **not**(F_2)— dependiendo de si está libre o ligada en F_1 o F_2 (si ocurre en cualquiera de ellas). Adviértase que en una fórmula de la forma $F = (F_1$ **and** $F_2)$ o $(F_1$ **or** $F_2)$, una variable de tupla puede estar libre en F_1 y ligada en F_2, o viceversa. En este caso, una ocurrencia de la variable de tupla está ligada y la otra está libre en F.
- Todas las ocurrencias *libres* de una variable de tupla t en F están **ligadas** en una fórmula F' de la forma $F' = (\exists\, t)(F)$ o $F' = (\forall\, t)(F)$. La variable de tupla está ligada al cuantificador especificado en F'. Por ejemplo, considere las dos fórmulas:

  F_1 : d.NOMBRED='Investigación'
  F_2 : $(\exists\, t)$ (d.NÚMEROD=t.ND)

  La variable de tupla d está libre tanto en F_1 como en F_2, en tanto que t está ligada al cuantificador $\exists$ en F_2.

Ahora podemos dar las reglas 3 y 4, para continuar la definición de fórmula que acabamos de iniciar:

3. Si F es una fórmula, también lo es $(\exists\, t)(F)$, donde t es una variable de tupla. La fórmula $(\exists\, t)(F)$ es **TRUE** si la evaluación de la fórmula F resulta **TRUE** para *alguna* (por lo menos una) tupla asignada a ocurrencias libres de t en F; de lo contrario, $(\exists\, t)(F)$ es **FALSE**.
4. Si F es una fórmula, también lo es $(\forall\, t)(F)$, donde t es una variable de tupla. La fórmula $(\forall\, t)(F)$ es **TRUE** si la evaluación de la fórmula F resulta **TRUE** para *toda tupla* (en el universo) asignada a ocurrencias libres de t en F; de lo contrario, $(\forall\, t)(F)$ es **FALSE**.

El cuantificador $(\exists)$ se denomina cuantificador existencial porque una fórmula $(\exists\, t)(F)$ es **TRUE** si "existe" alguna tupla t que haga que F sea **TRUE**. En el caso del cuantificador universal, $(\forall\, t)(F)$ es **TRUE** si t se sustituye por cualquier posible tupla que se pueda asignar a ocurrencias libres de t en F y F es **TRUE** después de efectuarse *cualquiera de esas sustituciones*. Se denomina cuantificador universal porque todas las tuplas del "universo de" tuplas deben hacer que F sea **TRUE**.

8.1.3 *Ejemplos de consultas empleando el cuantificador existencial*

Usaremos muchas de las consultas que presentamos en el capítulo 6 para dar al lector una idea de cómo se especifican las mismas consultas en el álgebra y en el cálculo relacionales.

Observe que algunas consultas son más fáciles de especificar en el álgebra relacional que en el cálculo, y viceversa.

CONSULTA 1
Obtener el nombre y la dirección de todos los empleados que trabajan para el departamento 'Investigación'.

C1 : {*t*.NOMBREP, *t*.APELLIDO, *t*.DIRECCIÓN | EMPLEADO(*t*) **and** ((∃*d*) (DEPARTAMENTO(*d*) **and** *d*.NOMBRED='Investigación' **and** *d*.NÚMEROD=*t*.ND))}

Las *únicas variables de tupla libres* en una expresión del cálculo relacional deberán ser las que aparezcan a la izquierda de la barra (|). En C1, t es la única variable libre, y después se *liga sucesivamente* a cada una de las tuplas que *satisfacen las condiciones* especificadas en C1, y se obtienen los atributos NOMBREP, APELLIDO y DIRECCIÓN de cada una de esas tuplas. Las condiciones EMPLEADO(t) y DEPARTAMENTO(d) especifican las relaciones de intervalo para t y d. La condición d.NOMBRED = 'Investigación' es una **condición de selección** y corresponde a una operación SELECCIONAR del álgebra relacional, en tanto que la condición d.NÚMEROD = t.ND es una **condición de reunión** y cumple un cometido similar al de la operación REUNIÓN (véase el capítulo 6).

CONSULTA 2
Para cada proyecto ubicado en 'Santiago', obtener una lista con el número de proyecto, el número del departamento que lo controla, y el apellido, la dirección y la fecha de nacimiento del gerente de dicho departamento.

C2 : {*p*.NÚMEROP, *p*.NÚMD, *m*.APELLIDO, *m*.FECHAN, *m*.DIRECCIÓN | PROYECTO(*p*) **and** EMPLEADO(*m*) **and** *p*.LUGARP='Santiago' **and** ((∃*d*)(DEPARTAMENTO(*d*) **and** *p*.NÚMD=*d*.NÚMEROD **and** *d*.NSSGTE=*m*.NSS))}

En C2 hay dos variables de tupla libres, p y m. La variable de tupla d está ligada al cuantificador existencial. La condición de consulta se evalúa para cada combinación de tuplas asignada a p y a m; y de todas las posibles combinaciones de tuplas a las que están ligadas p y m, sólo se seleccionan las combinaciones que satisfagan la condición.

Dos o más variables de tupla de una consulta pueden abarcar la misma relación. Por ejemplo, para especificar la consulta C8 —para cada empleado, obtener el nombre de pila y el apellido del empleado y el nombre de pila y el apellido de su supervisor inmediato— especificaremos dos variables de tupla e y s que abarquen la relación EMPLEADO:

C8 : {*e*.NOMBREP, *e*.APELLIDO, *s*.NOMBREP, *s*.APELLIDO | EMPLEADO(*e*) **and** EMPLEADO(*s*) **and** *e*.NSSSUPER=*s*.NSS}

CONSULTA 3'
Buscar los nombres de los empleados que trabajan en *algún* proyecto controlado por el departamento número 5. Ésta es una variación de la consulta 3 en la que "todos los" se cambia por "algún". En este caso necesitamos dos condiciones de reunión y dos cuantificadores existenciales.

C3' : {*e*.APELLIDO, *e*.NOMBREP | EMPLEADO(*e*) **and** ((∃ *x*)(∃ *w*) (PROYECTO(*x*) **and** TRABAJA_EN(*w*) **and** *x*.NÚMD=5 **and** *w*.NSSE=*e*.NSS **and** *x*.NÚMEROP=*w*.NÚMP))}

CONSULTA 4

Preparar una lista de números de los proyectos en los que intervenga un empleado cuyo apellido es 'Silva', ya sea como trabajador o como gerente del departamento que controla el proyecto.

C4 : {*p*.NÚMEROP | PROYECTO(*p*) **and**
(((∃ *e*)(∃ *w*)(EMPLEADO(*e*) **and** TRABAJA_EN(*w*) **and**
w.NÚMP=*p*.NÚMEROP **and** *e*.APELLIDO='Silva' **and** *e*.NSS=*w*.NSSE))
or
((∃ *m*)(∃ *d*)(EMPLEADO(*m*) **and** DEPARTAMENTO(*d*) **and**
p.NÚMD=*d*.NÚMEROD **and** *d*.NSSGTE=*m*.NSS **and** *m*.APELLIDO='Silva'))) }

Compare esto con la versión de esta consulta en álgebra relacional que se dio en el capítulo 6. La operación UNIÓN del álgebra relacional casi siempre puede sustituirse por una conectiva **or** en el cálculo relacional. En la siguiente sección estudiaremos la relación entre los cuantificadores universal y existencial y mostraremos la forma de convertir uno en el otro.

8.1.4 *Transformación entre los cuantificadores universal y existencial*

Ahora presentaremos algunas transformaciones bastante conocidas de la lógica matemática que expresan las relaciones que guardan entre sí los cuantificadores universal y existencial. Es posible transformar un cuantificador universal en un cuantificador existencial, y viceversa, y obtener una expresión equivalente. Podemos hacer una descripción informal de una transformación general como sigue: se puede transformar un tipo de cuantificador en el otro con la negación (anteponiendo **not**); **and** reemplaza a **or** y viceversa; una fórmula negada se transforma en no negada, y una fórmula no negada se convierte en negada. Algunos casos especiales de esta transformación pueden formularse como se muestra en seguida, donde ∄ denota **not**(∃):

(∀ *x*) (*P*(*x*)) ≡ (∄ *x*) (**not** (*P*(*x*)))
(∃ *x*) (*P*(*x*)) ≡ **not** (∀ *x*) (**not** (*P*(*x*)))
(∀ *x*) (*P*(*x*) **and** *Q*(*x*)) ≡ (∄ *x*) (**not** (*P*(*x*)) **or not** (*Q*(*x*)))
(∀ *x*) (*P*(*x*) **or** *Q*(*x*)) ≡ (∄ *x*) (**not** (*P*(*x*)) **and not** (*Q*(*x*)))
(∃ *x*) (*P*(*x*) **or** *Q*(*x*)) ≡ **not** (∀ *x*) (**not** (*P*(*x*)) **and not** (*Q*(*x*)))
(∃ *x*) (*P*(*x*) **and** *Q*(*x*)) ≡ **not** (∀ *x*) (**not** (*P*(*x*)) **or not** (*Q*(*x*)))

Observe además que se cumple lo siguiente, donde el símbolo ⇒ significa **implica**:

(∀ *x*) (*P*(*x*)) ⇒ (∃ *x*) (*P*(*x*))
(∄ *x*) (*P*(*x*)) ⇒ **not** (∀ *x*) (*P*(*x*))

Sin embargo, lo siguiente *no es verdadero:*

not (∀ *x*) (*P*(*x*)) ⇒ (∄ *x*) (*P*(*x*))

8.1.5 *Cómo usar el cuantificador universal*

Siempre que usemos un cuantificador universal, conviene seguir ciertas reglas para garantizar que nuestra expresión tenga sentido. Analizaremos estas reglas en relación con la consulta 3.

CONSULTA 3

Buscar los nombres de los empleados que trabajan en *todos* los proyectos controlados por el departamento número 5. Una forma de especificar esta consulta es utilizando el cuantificador universal, como se muestra.

C3 : {*e*.APELLIDO, *e*.NOMBREP | EMPLEADO(*e*) **and** ((∀ *x*)(**not** (PROYECTO(*x*))
or (**not** (*x*.NÚMD=5) **or**
((∃ *w*)(TRABAJA_EN(*w*) **and** *w*.NSSE=*e*.NSS **and** *x*.NÚMEROP=*w*.NÚMP)))))}

Podemos dividir C3 en sus componentes básicos como sigue:

C3 : {*e*.APELLIDO, *e*.NOMBREP | EMPLEADO(*e*) **and** *F*'}
F' = (∀ *x*)(**not** (PROYECTO(*x*)) **or** F_1)
F_1 = (**not** (*x*.NÚMD=5) **or** F_2)
F_2 = (∃ *w*)(TRABAJA_EN(*w*) **and** *w*.NSSE=*e*.NSS **and** *x*.NÚMEROP=*w*.NÚMP)

El truco consiste en excluir de la cuantificación universal todas las tuplas que no nos interesan, lo que podemos lograr haciendo que la condición sea TRUE *para todas esas tuplas*. Esto es necesario porque una variable de tupla abarcada por una cuantificación universal, como *x* en C3, debe resultar TRUE *para toda tupla posible* que se le asigne. Las primeras tuplas que se excluyen son las que no están en la relación *R* que nos interesa; luego se excluyen las tuplas de *R* que no nos interesan; por último, especificamos una condición F_2 que deben cumplir todas las tuplas de *R* restantes. En C3, *R* es la relación PROYECTO. Se seleccionan para el resultado de la consulta todas las tuplas *e* que hagan que F_2 sea TRUE *para todas las tuplas de* PROYECTO *restantes que no se han excluido*. Así pues, podemos explicar C3 como sigue:

1. Para que la fórmula *F*' = (∀ *x*)(*F*) sea TRUE, la fórmula *F* debe ser TRUE *para todas las tuplas del universo que se puedan asignar a x*. Sin embargo, en C3 sólo nos interesa que *F* sea TRUE para todas las tuplas de la relación PROYECTO que están bajo el control del departamento 5. Por tanto, la fórmula *F* tiene la forma (**not**(PROYECTO(*x*)) **or** F_1). La condición '**not**(PROYECTO(*x*)) **or**...' es TRUE para todas las tuplas *que no están en la relación* PROYECTO, y tiene el efecto de excluir estas tuplas de la consideración del valor lógico de F_1. Para *cada tupla* de la relación PROYECTO, F_1 debe ser TRUE para que *F* sea TRUE.
2. Siguiendo el mismo razonamiento, no queremos considerar las tuplas de la relación PROYECTO que no nos interesan. Como queremos que una tupla EMPLEADO seleccionada trabaje en todos los proyectos controlados por el departamento número 5, sólo nos interesan las tuplas PROYECTO en las que NÚMD = 5. Por tanto, podemos decir

   si (*x*.NÚMD=5) **entonces** F_2

   lo que equivale a

   (**not**(*x*.NÚMD=5) **or** F_2)

Así pues, la fórmula F_1 tiene la forma (PROYECTO(x) **and** (**not**(x.NÚMD=5) **or** F_2)). En el contexto de C3, esto significa que, para una tupla x de la relación PROYECTO, o bien NÚMD ≠ 5 o bien debe satisfacerse F_2.

3. Por último, F_2 proporciona la condición que queremos que cumpla una tupla EMPLEADO seleccionada: que el empleado trabaje en *todas las tuplas PROYECTO que todavía no se hayan excluido*. La consulta selecciona estas tuplas EMPLEADO.

En español, C3 proporciona la siguiente condición para seleccionar una tupla EMPLEADO e: para toda tupla x de la relación PROYECTO con x.NÚMD = 5, debe existir una tupla w en TRABAJA_EN tal que w.NSSE = e.NSS y w.NÚMP = x.NÚMEROP. Esto es lo mismo que decir que el EMPLEADO e trabaja en todo PROYECTO x en el DEPARTAMENTO número 5. (Esto ya suena familiar, ¿no es así?)

Si usamos la transformación general de cuantificador universal a existencial dada en la sección 8.1.4, podremos reformular la consulta C3 como se muestra en C3A:

C3A : {e.APELLIDO, e.NOMBREP | EMPLEADO(e) **and** (**not** (∃ x) (PROYECTO(x) **and** (x.NÚMD=5) **and**
(**not**(∃ w)(TRABAJA_EN(w) **and** w.NSSE=e.NSS **and** x.NÚMEROP=w.NÚMP)))))}

Veamos ahora algunos ejemplos adicionales de consultas que utilizan cuantificadores.

CONSULTA 6
Obtener los nombres de los empleados que no tienen dependientes.

C6 : {e.NOMBREP, e.APELLIDO | EMPLEADO(e) **and** (**not**(∃d)(DEPENDIENTE (d) **and** e.NSS=d.NSSE))}

Por la regla de transformación general, podemos reformular C6 como sigue:

C6A : {e.NOMBREP, e.APELLIDO | EMPLEADO(e) **and** ((∀ d)(**not**(DEPENDIENTE(d)) **or not**(e.NSS=d.NSSE)))}

CONSULTA 7
Listar los nombres de los gerentes que tienen por lo menos un dependiente.

C7 : {e.NOMBREP, e.APELLIDO | EMPLEADO(e) **and** ((∃ d) (∃ p) (DEPARTAMENTO(d) **and** DEPENDIENTE(p) **and** e.NSS=d.NSSGTE **and** p.NSSE=e.NSS))}

8.1.6 *Expresiones seguras*

Siempre que usemos cuantificadores universales, cuantificadores existenciales o negación de predicados en una expresión del cálculo, deberemos asegurarnos de que la expresión resultante tenga sentido. Una **expresión segura** en cálculo relacional es una que siempre produce un *número finito de tuplas* como resultado; de lo contrario, se dice que la expresión es **insegura**. Por ejemplo, la expresión

{t | **not**(EMPLEADO(t))}

es *insegura* porque produce todas las tuplas del universo que *no son* tuplas EMPLEADO: un número infinito de tuplas. Si seguimos las reglas que dimos al examinar C3, obtendremos una expresión segura si utilizamos cuantificadores universales. Podemos definir las "expresiones seguras" de manera más precisa presentando el concepto de *dominio de una expresión del cálculo relacional de tuplas*: el conjunto de todos los valores que aparecen como valores constantes en la expresión o bien que existen en cualquier tupla de las relaciones a las que se hace referencia en la expresión. El dominio de {t | **not**(EMPLEADO(t))} es el conjunto de todos los valores de atributos que aparecen en alguna tupla de la relación EMPLEADO (para cualquier atributo). El dominio de la expresión C3A incluiría todos los valores que aparecen en EMPLEADO, PROYECTO y TRABAJA_EN (en UNIÓN con el valor 5 que aparece en la consulta misma).

Se dice que una expresión es **segura** si todos los valores de su resultado pertenecen al dominio de la expresión. Observe que el resultado de {t | **not**(EMPLEADO(t))} es inseguro porque, en general, incluirá tuplas (y por tanto valores) que no están en la relación EMPLEADO; tales valores no pertenecen al dominio de la expresión. Todos los demás ejemplos que dimos son expresiones seguras.

8.1.7 *Cuantificadores en SQL*

La función EXISTS de SQL es similar al cuantificador existencial del cálculo relacional. Cuando escribimos

```
SELECT   ...
FROM     ...
WHERE    EXISTS  (SELECT  *
                  FROM    R X
                  WHERE   P(X) )
```

en SQL, ello equivale a decir que una variable de tupla X que abarca la relación R está cuantificada existencialmente. La consulta anidada a la que se aplica la función EXISTS normalmente está correlacionada con la consulta exterior; esto es, la condición $P(X)$ incluye algún atributo de las relaciones de la consulta exterior. La condición WHERE de la consulta exterior resulta TRUE si la consulta anidada devuelve un resultado no vacío que contenga una o más tuplas.

SQL no cuenta con un cuantificador universal. El empleo de un cuantificador existencial negado (∃ x) especificado con NOT EXISTS es la forma en que SQL maneja la cuantificación universal, como se ilustró con C3 en el capítulo 7.

8.2 El lenguaje QUEL*

QUEL es un lenguaje de definición y manipulación de datos que se creó originalmente para el SGBD relacional INGRES, de uso muy extendido. La primera versión de INGRES, acrónimo de *Interactive Graphics and Retrieval System* (sistema interactivo de gráficos y obtención de datos), se creó como proyecto de investigación en la University of California en Berkeley a mediados de los años setenta; a ésta se le conoce como INGRES universitario. Un SGBD comercial

de INGRES ha estado en el mercado desde 1980. QUEL se puede usar como lenguaje de consulta interactivo o incorporarse en un lenguaje de programación anfitrión. Incluye una gama de funcionalidad similar a la de las primeras versiones de SQL. Nuestra exposición seguirá un patrón semejante al empleado en la presentación del SQL en el capítulo 7, pero en forma resumida. Los aspectos de consulta de QUEL están íntimamente relacionados con el cálculo relacional de tuplas que vimos en la sección anterior, pero ampliados con las funciones agregadas y la agrupación.

8.2.1 *Definición de datos y de almacenamiento en QUEL*

La orden CREATE y los tipos de datos de QUEL. La orden CREATE (crear) sirve para especificar una **relación base** —una relación cuyas tuplas están almacenadas físicamente en la base de datos— y sus atributos. Se dispone de varios tipos de datos para los atributos. Entre los tipos de datos numéricos están I1, I2 e I4 para enteros de uno, dos y cuatro bytes de longitud, y F4 y F8 para números reales de punto flotante de 4 y 8 bytes de longitud. Los tipos de datos de cadenas de caracteres incluyen C*n* y CHAR(*n*) para cadenas de longitud fija de *n* caracteres y TEXT(*n*) y VARCHAR(*n*) para cadenas de longitud variable máxima de *n* caracteres.† Además, se cuenta con un tipo de datos DATE que ofrece formatos flexibles, entre ellos valores absolutos fecha/hora como '19-OCT-1920 12:00 pm' o valores relativos como '2 years 4 months' (2 años 4 meses) o '1 day 4 hours 30 minutes' (1 día 4 horas 30 minutos). También está disponible un tipo de datos MONEY (dinero), que es un número de 16 dígitos en el que los dos del extremo derecho representan centavos. La figura 8.1 muestra cómo se podrían declarar en QUEL las relaciones EMPLEADO y DEPARTAMENTO de la figura 6.5.

La orden INDEX. La orden INDEX sirve para especificar un índice (sólo de primer nivel) para una relación; de hecho, los índices se tratan como relaciones base ordenadas cuyos únicos atributos son los atributos de indización, y el SGBD mantiene apuntadores a los registros correspondientes. En nuestra terminología del capítulo 5, es un *índice secundario de un solo nivel*. Se puede crear un índice de múltiples niveles especificando un B-TREE (árbol B) sobre un índice ya existente.

```
CREATE EMPLEADO      ( NOMBREP        =  TEXT(15),
                       INIC           =  C1,
                       APELLIDO       =  TEXT(15),
                       NSS            =  C9,
                       FECHAN         =  DATE,
                       DIRECCIÓN      =  TEXT(30),
                       SEXO           =  C1,
                       SALARIO        =  MONEY,
                       NSSSUPER       =  C9,
                       ND             =  I4 );
CREATE DEPARTAMENTO  ( NOMBRED        =  TEXT(15),
                       NÚMEROD        =  I4,
                       NSSGTE         =  C9,
                       FECHAINICGTE   =  DATE );
```

Figura 8.1 Declaración de las relaciones EMPLEADO y DEPARTAMENTO de la figura 6.5 en QUEL.

†C*n* se usó en el INGRES universitario y se comporta de manera peculiar cuando se aplican comparaciones de cadenas. TEXT(*n*) se introdujo en el INGRES comercial y da resultados más predecibles.

La orden MODIFY y empleo de UNIQUE. Si queremos especificar o modificar la estructura de almacenamiento de una relación base o de un índice, usamos la orden MODIFY. Las estructuras de almacenamiento disponibles son ISAM (indizada secuencial), HASH (dispersión), BTREE (árbol B), HEAP (archivo no ordenado) y HEAPSORT (ordenar los registros ahora, pero sin mantener el orden cuando se inserten nuevas tuplas). La letra C antepuesta a cualquier nombre de estructura de almacenamiento —por ejemplo, CHASH o CBTREE— indica que los archivos y sus caminos de acceso se almacenarán en forma comprimida, ahorrando espacio pero aumentando los tiempos de obtención de datos y actualización. Cada relación base (o índice) puede tener cuando más una estructura de almacenamiento definida para ella. La estructura de almacenamiento por omisión usual para una relación base es HEAP.

Los atributos clave se especifican incluyendo la palabra reservada UNIQUE (única) en una orden MODIFY; así pues, al igual que en las primeras versiones de SQL, no es posible especificar atributos clave independientemente de una estructura de almacenamiento específica para la relación. A fin de ilustrar el empleo de MODIFY, suponga que queremos especificar lo siguiente: una clave y un árbol B sobre el atributo NSS de EMPLEADO, un índice sobre la combinación de los atributos APELLIDO y NOMBREP de EMPLEADO, una clave y un acceso por dispersión sobre el atributo NOMBRED de DEPARTAMENTO, y una clave y un índice de árbol B comprimido sobre el atributo NÚMEROD de DEPARTAMENTO. Las instrucciones de L1 realizan todas estas tareas:

```
L1:  MODIFY EMPLEADO TO BTREE UNIQUE ON NSS;
     INDEX ON EMPLEADO IS ÍNDICE_NOMBRE (APELLIDO, NOMBREP);
     MODIFY DEPARTAMENTO TO HASH UNIQUE ON NOMBRED;
     INDEX ON DEPARTAMENTO IS ÍNDICE_NÚMEROD (NÚMEROD);
     MODIFY ÍNDICE_NÚMEROD TO CBTREE UNIQUE ON NÚMEROD;
```

Si ya no se necesita una relación base o un índice, podemos eliminarlos con la orden DESTROY (destruir). En L2 damos un ejemplo:

```
L2:  DESTROY ÍNDICE_NÚMEROD, ÍNDICE_NOMBRE;
```

El QUEL original no tiene órdenes para añadir atributos a las tablas (ALTER en SQL), ni valores NULL explícitos como SQL. Un valor faltante se representa con un espacio en blanco en el caso de tipos de datos de cadena y con 0 (cero) en el caso de tipos de datos numéricos.

8.2.2 *Panorama de las construcciones de consulta de QUEL*

En QUEL las consultas básicas de obtención de datos del tipo **seleccionar–proyectar–reunir** son muy similares al cálculo relacional de tuplas. Se utilizan dos cláusulas, RETRIEVE (obtener) y WHERE (donde); RETRIEVE especifica los atributos que se desea obtener —los atributos de proyección— y WHERE especifica las condiciones de selección y de reunión. QUEL no tiene una cláusula FROM como SQL; en una consulta QUEL, todos los atributos *se deben calificar explícitamente*, sea con el nombre de su relación o con una variable de tupla declarada para abarcar su relación. Las variables de tupla se declaran explícitamente con el enunciado RANGE (intervalo) de QUEL.

CONSULTA 0
Obtener la fecha de nacimiento y la dirección del empleado cuyo nombre es 'José B. Silva'.

```
C0:  RETRIEVE  (EMPLEADO.FECHAN, EMPLEADO.DIRECCIÓN)
     WHERE     EMPLEADO.NOMBREP='José' AND EMPLEADO.INIC='B'
               AND EMPLEADO.APELLIDO='Silva'
```

Como alternativa, podemos especificar variables de tupla en declaraciones RANGE y utilizarlas en la consulta. En los ejemplos restantes, supondremos que se han declarado las variables RANGE de L3:

```
L3:  RANGE OF  E, S IS EMPLEADO,
               D IS DEPARTAMENTO,
               P IS PROYECTO,
               T IS TRABAJA_EN,
               DEP IS DEPENDIENTE,
               L IS LUGARES_DEPTOS
```

CONSULTA 1
Obtener el nombre y la dirección de todos los empleados que pertenecen al departamento 'Investigación'.

```
C1:  RETRIEVE  (E.NOMBREP, E.APELLIDO, E.DIRECCIÓN)
     WHERE     D.NOMBRED='Investigación' AND D.NÚMEROD=E.ND
```

CONSULTA 2
Para cada proyecto ubicado en 'Santiago', listar el número del proyecto, el número del departamento controlador, y el apellido, la dirección y la fecha de nacimiento del gerente de ese departamento.

```
C2:  RETRIEVE  (P.NÚMEROP, P.NÚMD, E.APELLIDO, E.FECHAN,
               E.DIRECCIÓN)
     WHERE     P.NÚMD=D.NÚMEROD AND D.NSSGTE=E.NSS AND
               P.LUGARP='Santiago'
```

CONSULTA 8
Para cada empleado, obtener su nombre de pila y apellido y el nombre de pila y apellido de su supervisor inmediato.

```
C8:  RETRIEVE  (E.NOMBREP, E.APELLIDO, S.NOMBREP, S.APELLIDO)
     WHERE     E.NSSSUPER=S.NSS
```

Si queremos obtener *todos los atributos* de las tuplas seleccionadas, usamos la palabra reservada ALL (todos). La consulta C1D obtiene todos los atributos de un EMPLEADO y del DEPARTAMENTO al que pertenece, para todos los empleados del departamento 'Investigación':

```
C1D: RETRIEVE  (E.ALL, D.ALL)
     WHERE     D.NOMBRED='Investigación' AND E.ND=D.NÚMEROD
```

Al igual que en SQL, los resultados de una consulta en QUEL pueden tener tuplas repetidas. Para eliminarlas especificamos la palabra reservada UNIQUE en la cláusula RETRIEVE (esto es similar a DISTINCT en SQL). La consulta 11 obtiene todos los salarios de los empleados, conservando los duplicados, mientras que C11A elimina los salarios repetidos del resultado de la consulta:

```
C11:  RETRIEVE  (E.SALARIO)
C11A: RETRIEVE  UNIQUE (E.SALARIO)
```


CONSULTA 3'
Obtener los nombres de los empleados que trabajan en *algún* proyecto controlado por el departamento 5.

```
C3': RETRIEVE  UNIQUE (E.NOMBREP, E.APELLIDO)
     WHERE     P.NÚMD=5 AND P.NÚMEROP=T.NÚMP AND T.NSSE=E.NSS
```

En QUEL, toda variable de intervalo que aparezca en la cláusula WHERE de una consulta y que no aparezca en la cláusula RETRIEVE estará *cuantificada implícitamente por un cuantificador existencial*. Las variables de tupla P y T están cuantificadas implícitamente por el cuantificador existencial en C3'. En el caso de consultas que impliquen cuantificadores universales o cuantificadores existenciales negados, debemos usar la función COUNT (cuenta) o bien la función ANY (cualquier). QUEL cuenta con las funciones integradas COUNT, SUM, MIN, MAX y AVG. Las funciones adicionales COUNTU, SUMU y AVGU *eliminan tuplas repetidas antes de aplicar las funciones* COUNT, SUM y AVG.

Todas las funciones QUEL se pueden usar ya sea en la cláusula RETRIEVE o en la cláusula WHERE. Siempre que se utilice una función en la cláusula RETRIEVE habrá que darle un *nombre de atributo independiente*, que aparecerá como cabecera de columna en el resultado de la consulta. En C15, la relación resultante tendrá los nombres de columna SUMSAL, SALMÁX, SALMÍN y SALMEDIO.

CONSULTA 15
Obtener la suma de los salarios de todos los empleados, el salario máximo, el salario mínimo y el salario medio.

```
C15: RETRIEVE  (SUMSAL = SUM (E.SALARIO), SALMÁX = MAX (E.SALARIO),
               SALMÍN = MIN (E.SALARIO), SALMEDIO = AVG (E.SALARIO))
```

CONSULTAS 17 Y 18
Obtener el número total de empleados de la compañía (C17) y el número de empleados del departamento 'Investigación' (C18).

```
C17: RETRIEVE  (TOTAL_EMPS = COUNT (E.NSS))
C18: RETRIEVE  (EMPS_INVEST = COUNT (E.NSS WHERE E.ND=D.
               NÚMEROD AND D.NOMBRED='Investigación'))
```

En QUEL se puede usar el calificador BY (por) en cualquier especificación de función para indicar una cierta agrupación de tuplas; así, se pueden usar diferentes agrupaciones en la misma consulta. Cada función puede tener sus propios atributos de agrupación, así como sus propias condiciones WHERE, y la agrupación puede usarse en la cláusula RETRIEVE o en la cláusula WHERE.

CONSULTA 20
Para cada departamento, obtener el número de departamento, el número de empleados del departamento y su salario medio.

```
C20: RETRIEVE  (E.ND, NÚM_DE_EMPS = COUNT (E.NSS BY E.ND),
               SAL_MEDIO = AVG (E.SALARIO BY E.ND))
```

C20 agrupa tuplas EMPLEADO por número de departamento al especificar "BY E.ND". La cuenta de los empleados por departamento se escribe COUNT(E.NSS BY E.ND). El atributo de agrupación E.ND debe aparecer también de manera independiente en la lista RETRIEVE para que la consulta tenga sentido.

CONSULTA 21
Para cada proyecto, obtener el número de proyecto, su nombre y el número de empleados que trabajan en ese proyecto.

```
C21: RETRIEVE (T.NÚMP, P.NOMBREPR, NÚM_DE_EMPS = COUNT (T.NSSE
                BY T.NÚMP, P.NOMBREPR WHERE T.NÚMP=P.NÚMEROP))
```

C21 ilustra un ejemplo de agrupación en el que dos relaciones se reúnen con base en la condición T.NÚMP = P.NÚMEROP y luego se agrupan las tuplas por (T.NÚMP, P.NOMBREPR), calculándose entonces el resultado de la consulta. C21 ilustra el empleo tanto de la agrupación como de una condición WHERE *dentro de la especificación de una función agregada*. Esto permite aplicar varias funciones a diferentes conjuntos de tuplas dentro de la misma consulta.

CONSULTA 22
Para cada proyecto en el que trabajen más de dos empleados, obtener el número del proyecto y el número de empleados que trabajan en él.

```
C22: RETRIEVE (T.NÚMP, NÚM_DE_EMPS = COUNT (T.NSSE BY T.NÚMP))
     WHERE    COUNT (T.NSSE BY T.NÚMP) > 2
```

Otra función de QUEL, ANY, sirve para especificar una cuantificación existencial explícita. ANY se aplica a una subconsulta; si el resultado incluye por lo menos una tupla, ANY devuelve 1; en caso contrario, devuelve 0. Así pues, ANY es un tanto parecida a la función EXISTS de SQL, excepto que EXISTS devuelve TRUE o FALSE en vez de 1 o 0. ANY suele aplicarse a una subconsulta anidada que usa agrupación o a una condición WHERE.

CONSULTA 12
Obtener el nombre de todos los empleados que tienen un dependiente con los mismos nombres de pila y sexo que el empleado.

```
C12: RETRIEVE (E.NOMBREP, E.APELLIDO)
     WHERE ANY (DEP.NOMBRE_DEPENDIENTE BY E.NSS
     WHERE E.NSS=DEP.NSSE
     AND E.NOMBREP=DEP.NOMBRE_DEPENDIENTE AND
     E.SEXO=DEP.SEXO) = 1
```

CONSULTA 6
Obtener los nombres de los empleados que no tienen dependientes.

```
C6:  RETRIEVE (E.NOMBREP, E.APELLIDO)
     WHERE    ANY    (DEP.NOMBRE_DEPENDIENTE BY E.NSS
                     WHERE E.NSS=DEP.NSSE) = 0
```

CONSULTA 3
Obtener los números de seguro social de los empleados que trabajan en *todos* los proyectos controlados por el departamento número 5.

```
C3:  RETRIEVE (E.NSS)
     WHERE    ANY  (P.NÚMEROP
                    WHERE (P.NÚMD=5)
                          AND
                          ANY (T.NÚMP BY E.NSS
                    WHERE E.NSS=T.NSSE AND
                    P.NÚMEROP=T.NÚMP) =0) =0
```


QUEL permite efectuar comparaciones parciales de cadenas mediante dos caracteres reservados: "*" reemplaza un número arbitrario de caracteres, y "?" reemplaza un solo carácter arbitrario. Por ejemplo, si queremos obtener todos los empleados cuya dirección está en Santiago, Estado de México, podemos usar C25.

```
C25: RETRIEVE (E.APELLIDO, E.NOMBREP)
     WHERE    E.DIRECCIÓN = '*Santiago, MX*'
```

QUEL permite el empleo de los operadores aritméticos estándar "+", "–", "*"y "/", así como la colocación del resultado de una consulta *en una relación*; esto se hace con la palabra reservada INTO. Por ejemplo, si queremos conservar el resultado de C27 en una relación llamada AUMENTOS_PRODUCTOX que también contenga el salario actual en cada tupla, además de una columna llamada PROY cuyo valor sea 'ProductoX' para todas las tuplas, podemos usar C27A:

```
C27A: RETRIEVE INTO AUMENTOS_PRODUCTOX (PROY = 'ProductoX',
                    E.NOMBREP,
                    E.APELLIDO, SALARIO_ACTUAL = E.SALARIO,
                    SALARIO_PROPUESTO = E.SALARIO * 1.1)
      WHERE         E.NSS=T.NSSE AND T.NÚMP=P.NÚMEROP AND
                    P.NOMBREPR=
                    'ProductoX'
```

QUEL también tiene una cláusula SORT BY (ordenar por) similar a la de SQL. En QUEL podemos usar tres instrucciones para modificar la base de datos: APPEND, DELETE y REPLACE. El lenguaje cuenta también con recursos para definir vistas, además de un mecanismo para incorporar QUEL en un lenguaje de programación (llamado EQUEL). Hagamos un breve análisis de las actualizaciones en QUEL. Para insertar una tupla nueva en una relación, QUEL cuenta con la orden APPEND (anexar). Por ejemplo, si queremos añadir una nueva tupla a la relación EMPLEADO de la figura 6.6, podemos usar A1:

```
A1:  APPEND TO EMPLEADO (NOMBREP = 'Ricardo', INIC = 'C', APELLIDO =
                'Martínez', NSS = '653298653', FECHAN = '30-DIC-52',
                DIRECCIÓN = 'Olmo 98, Cedros, MX', SEXO = 'M',
                SALARIO = 37000, NSSSUPER = '987654321', ND = 4)
```

La orden APPEND también permite insertar múltiples tuplas en una relación seleccionando el resultado de otra consulta. La orden DELETE sirve para eliminar tuplas de una relación, e incluye una cláusula WHERE para seleccionar las tuplas que se van a eliminar:

```
A4A: DELETE   EMPLEADO
     WHERE    E.APELLIDO='Bojórquez'
A4B: DELETE   EMPLEADO
     WHERE    E.NSS='123456789'
A4C: DELETE   EMPLEADO
     WHERE    E.ND=D.NÚMEROD AND D.NOMBRED='Investigación'
```

La orden REPLACE (reemplazar) sirve para modificar los valores de los atributos. Una cláusula WHERE selecciona de una sola relación las tuplas que se van a modificar. Por ejemplo,

si queremos cambiar el lugar y el departamento controlador del proyecto 10 a 'Belén' y departamento 5, respectivamente, usamos A5:

```
A5:  REPLACE PROYECTO(LUGARP = 'Belén', NÚMD = 5)
     WHERE   P.NÚMEROP=10
```

8.2.3 *Comparación entre QUEL y SQL*

En esta sección haremos una breve comparación entre QUEL y SQL. Vale la pena destacar los siguientes puntos:

- Tanto QUEL como SQL se basan en variaciones del cálculo relacional de tuplas; sin embargo, QUEL se acerca mucho más al cálculo relacional de tuplas que SQL.
- En SQL el anidamiento de bloques SELECT ... FROM ... WHERE ... se puede repetir arbitrariamente hasta cualquier número de niveles; en QUEL el anidamiento está restringido a un nivel.
- Tanto SQL como QUEL utilizan cuantificadores existenciales implícitos. Ambos manejan la cuantificación universal en términos de una cuantificación existencial equivalente.
- El mecanismo de agrupación de QUEL —la cláusula BY ... WHERE ...— puede ocurrir cualquier número de veces en las cláusulas RETRIEVE o WHERE, lo que permite diferentes agrupaciones en la misma consulta. En SQL sólo se permite una agrupación por consulta mediante la cláusula GROUP BY ... HAVING ..., así que es preciso anidar las consultas para efectuar diferentes agrupaciones.
- SQL permite algunas operaciones de conjuntos explícitas del álgebra relacional, como UNION; éstas no están disponibles en QUEL, pero se manejan especificando condiciones de selección y de reunión más complejas.
- QUEL permite especificar en un solo enunciado un archivo temporal para recibir la salida de una consulta; en SQL es preciso emplear una instrucción CREATE aparte para crear una tabla antes de poder hacer esto.
- El operador ANY devuelve un valor entero de 0 o 1 en QUEL; el operador EXISTS, que se usa para fines similares en SQL, devuelve un valor booleano de TRUE o FALSE.
- SQL ha evolucionado considerablemente, convirtiéndose en una norma *de facto* para las bases de datos relacionales.

8.3 Cálculo relacional de dominios*

El cálculo relacional de dominios, o simplemente **cálculo de dominios**, es el otro tipo de lenguaje formal para bases de datos relacionales basado en el cálculo de predicados. Sólo existe un lenguaje de consulta en el mercado, QBE (véase la Sec. 8.4), que guarda cierta relación con el cálculo de dominios, aunque QBE se creó antes de la especificación formal del cálculo mencionado.

El cálculo de dominios difiere del cálculo de tuplas en el *tipo de variables* empleadas en las fórmulas: en vez de que las variables abarquen tuplas, abarcan valores individuales de

los dominios de atributos. Si queremos formar una relación de grado n como resultado de una consulta, deberemos tener n de estas **variables de dominio**: una para cada atributo. Una expresión del cálculo de dominios tiene la forma

$$\{x_1, x_2, ..., x_n \mid \text{COND}\ (x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m})\}$$

donde $x_1, x_2, ..., x_n, x_{n+1}, x_{n+2}, ..., x_{n+m}$ son variables de dominio que abarcan dominios (de atributos) y COND es una **condición** o **fórmula** del cálculo relacional de dominios. Las fórmulas se componen de **átomos**. Los átomos de una fórmula son un tanto diferentes de los del cálculo de tuplas y pueden ser cualquiera de los siguientes:

1. Un átomo de la forma $R(x_1, x_2, ..., x_j)$, donde R es el nombre de una relación de grado j y cada x_i, con $1 \leq i \leq j$, es una variable de dominio. Este átomo expresa que una lista de valores de $<x_1, x_2, ..., x_j>$ debe ser una tupla en la relación cuyo nombre es R, donde x_i es el valor del i-ésimo atributo de la tupla. A fin de hacer más concisas las expresiones del cálculo de dominio, *se omiten las comas* de la lista de variables; así pues, escribimos

   $\{x_1\ x_2 ...\ x_n \mid R(x_1\ x_2\ x_3)$ **and** $...\}$

   en lugar de:

   $\{x_1, x_2, ..., x_n \mid R(x_1, x_2, x_3)$ **and** $...\}$
2. Un átomo de la forma x_i **op** x_j, donde **op** es uno de los operadores de comparación del conjunto $\{=, \neq, <, \leq, >, \geq\}$ y x_i y x_j son variables de dominio.
3. Un átomo de la forma x_i **op** c o c **op** x_j, donde **op** es uno de los operadores de comparación del conjunto $\{=, \neq, <, \leq, >, \geq\}$, x_i y x_j son variables de dominio y c es un valor constante.

Al igual que en el cálculo de tuplas, la evaluación de un átomo resulta ya sea TRUE o FALSE para un conjunto específico de valores, y estos resultados se denominan **valores lógicos** de los átomos. En el caso 1, si se asignan a las variables de dominio valores que correspondan a una tupla de la relación especificada R, el átomo será TRUE. En los casos 2 y 3, si se asignan a las variables de dominio valores que satisfagan la condición, el átomo será TRUE.

De manera similar al cálculo relacional de tuplas, las fórmulas se componen de átomos, variables y cuantificadores, así que no repetiremos aquí las especificaciones de las fórmulas. A continuación daremos algunos ejemplos de consultas especificadas en el cálculo de dominios. Usaremos letras minúsculas $l, m, n, ..., x, y, z$ para las variables de dominio.

CONSULTA 0

Obtener la fecha de nacimiento y la dirección del empleado cuyo nombre es 'José B. Silva'.

C0: $\{uv \mid (\exists q)\ (\exists r)\ (\exists s)$
$(\text{EMPLEADO}(qrstuvwxyz)$ **and** q='José' **and** r='B' **and** s='Silva')$\}$

Necesitamos diez variables para la relación EMPLEADO, una para cubrir el dominio de cada atributo en orden. De las diez variables $q, r, s, ..., z$, sólo q, r y s están ligadas a un cuantificador existencial; las demás son libres. Primero especificamos los *atributos solicitados*, FECHAN y DIRECCIÓN, con las variables de dominio u para FECHAN y v para DIRECCIÓN. Luego, después de la barra (|), especificamos la condición para seleccionar una tupla: es decir, que

la secuencia de valores asignados a las variables *qrstuvwxyz* sea una tupla de la relación EMPLEADO y que los valores de *q* (NOMBREP), *r* (INIC) y *s* (APELLIDO) sean 'José', 'B' y 'Silva', respectivamente. Adviértase que la cuantificación existencial *sólo abarca a las variables que participan en una condición*.

Una notación alternativa para escribir esta consulta es asignar las constantes 'José', 'B' y 'Silva' directamente como se muestra en C0A, donde todas las variables son libres:

C0A: {*uv* | EMPLEADO('José', 'B', 'Silva', *t, u, v, w, x, y, z*)}

CONSULTA 1
Obtener el nombre y la dirección de todos los empleados que pertenecen al departamento 'Investigación'.

C1: {*qsv* | (∃ *z*) (EMPLEADO(*qrstuvwxyz*) **and** (∃ *l*) (∃ *m*)
(DEPARTAMENTO(*lmno*) **and** l='Investigación' **and** *m*=*z*))}

Una condición que relaciona dos variables de dominio que abarcan atributos de dos relaciones, como $m = z$ en C1, es una **condición de reunión**; por otro lado, una condición que relaciona una variable de dominio con una constante, como l = 'Investigación', es una **condición de selección**.

CONSULTA 2
Para cada proyecto ubicado en 'Santiago', listar el número del proyecto, el número del departamento controlador, y el apellido, la dirección y la fecha de nacimiento del gerente de ese departamento.

C2: {*iksuv* | (∃ *j*) (PROYECTO(*hijk*) **and** (∃ *t*) (EMPLEADO(*qrstuvwxyz*) **and**
(∃ *m*) (∃ *n*) (DEPARTAMENTO(*lmno*) **and** *k*=*m* **and** *n*=*t* **and**
j='Santiago'))))}

CONSULTA 6
Obtener los nombres de los empleados que no tienen dependientes.

C6:{*qs* | (∃ *t*) (EMPLEADO(*qrstuvwxyz*) **and** (**not**(∃ *l*) (DEPENDIENTE(*lmnop*)
and *t*=*l*))))}

La consulta 6 puede reexpresarse empleando cuantificadores universales en lugar de cuantificadores existenciales, como se muestra en C6A:

C6A:{*qs* | (∃ *t*) (EMPLEADO(*qrstuvwxyz*) **and** ((∀ *l*) (**not**(DEPENDIENTE(*lmnop*))
or not(*t*=*l*)))))}

CONSULTA 7
Listar los nombres de los gerentes que tienen por lo menos un dependiente.

C7:{*sq* | (∃ *t*) (EMPLEADO(*qrstuvwxyz*) **and** ((∃ *j*) (DEPARTAMENTO(*hijk*)
and ((∃ *l*) (DEPENDIENTE(*lmnop*) **and** *t*=*j* **and** *l*=*t*))))))}

Como ya mencionamos, es posible demostrar que cualquier consulta que se pueda formular en el álgebra relacional se puede expresar también en el cálculo relacional de dominios o en el de tuplas. Además, toda expresión segura en el cálculo de dominios o en el de tuplas puede expresarse en el álgebra relacional.

8.4 Panorama del lenguaje QBE*

QBE (*Query By Example*: consulta por ejemplo) es un lenguaje de consulta relacional creado en IBM Research que cualquier usuario puede emplear con facilidad. QBE es un producto de IBM que se encuentra en el mercado como parte de la opción de interfaz QMF (*Query Management Facility*: recurso de gestión de consultas) de DB2. Difiere de SQL y de QUEL en que el usuario no tiene que especificar explícitamente una consulta estructurada; más bien, la consulta se formula llenando **plantillas** de relaciones que se exhiben en la pantalla de una terminal. La figura 8.2 muestra el aspecto que podrían tener dichas plantillas en la base de datos de la figura 6.6. El usuario no tiene que recordar los nombres de los atributos ni de las relaciones, porque se exhiben como parte de las plantillas. Es más, el usuario no tiene que seguir reglas de sintaxis rígidas para especificar las consultas, pues las constantes y las variables se introducen en las columnas de las plantillas para construir un **ejemplo** relacionado con la solicitud de obtención de datos o de actualización. QBE está relacionado con el cálculo relacional de dominios, como habremos de ver, y se ha demostrado que su especificación original es relacionalmente completa.

Las consultas de obtención de datos se especifican llenando ciertas columnas de las plantillas de relaciones. Al introducir valores constantes en una plantilla, se teclean tal como son; pero también se pueden introducir **valores de ejemplo**, que van precedidos por el carácter "_" (subrayado). Se utiliza el prefijo "P." (de *Print*: imprimir) para indicar que se deben obtener los valores de una columna en particular. Por ejemplo, consideremos la consulta C0: obtener la fecha de nacimiento y la dirección de 'José B. Silva'; esto puede especificarse en QBE como se muestra en la figura 8.3(a).

En la figura 8.3(a) especificamos un ejemplo del *tipo de fila* que nos interesa. Los valores de ejemplo precedidos por "_" representan *variables de dominio libres* (véase la Sec. 8.3). Los valores constantes reales, como José, B. y Silva, sirven para seleccionar tuplas de la base de datos que tienen esos mismos valores. El resultado de la consulta incluirá los valores de todas las columnas en las que aparezca el prefijo P.

C0 se puede abreviar como en la figura 8.3(b). No hay necesidad de especificar valores de ejemplo para las columnas que no nos interesan. Es más, como los valores de ejemplo

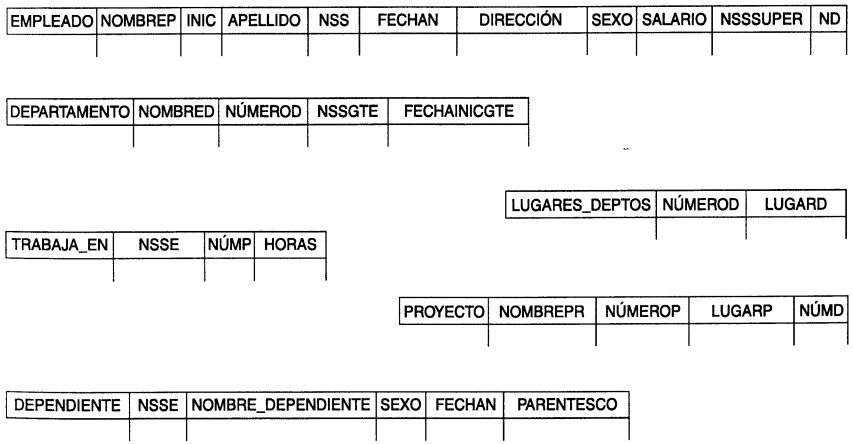


Figura 8.2 El esquema relacional de la figura 6.6, tal como podría presentarlo QBE.

son completamente arbitrarios, podemos limitarnos a especificar nombres de variables para ellos, como se aprecia en la figura 8.3(c). También podemos omitir por completo los valores de ejemplo, como se ilustra en la figura 8.3(d), y simplemente especificar P. en las columnas que se desea consultar.

A fin de percibir la semejanza entre las consultas QBE y el cálculo relacional de dominios, comparemos la figura 8.3(d) con C0 (simplificada) en el cálculo de dominios, a saber:

C0: { *uv* | EMPLEADO(*qrstuvwxyz*) **and** *q*='José' **and** *r*='B' **and** s='Silva'}

Podemos concebir cada columna de una plantilla QBE como una *variable de dominio implícita*; así pues, NOMBREP corresponde a la variable de dominio *q*, INIC corresponde a *r*, ..., y ND corresponde a *z*. En la consulta QBE, las columnas con P. corresponden a las variables especificadas a la izquierda de la barra (|) en el cálculo de dominios, y las columnas con valores constantes corresponderán a las variables para las que se especificaron condiciones de selección de igualdad. La condición EMPLEADO(*qrstuvwxyz*) y los cuantificadores existenciales están implícitos en la consulta QBE porque se utiliza la plantilla correspondiente a la relación EMPLEADO.

En QBE, la interfaz del usuario permite primero al usuario elegir las tablas (relaciones) que necesita para formular una consulta exhibiendo una lista de todos los nombres de relaciones. A continuación se muestran las plantillas de las relaciones elegidas. El usuario se coloca en las columnas apropiadas de las plantillas y especifica la consulta. Se utilizan teclas de función especiales para avanzar a la siguiente columna o a la anterior de la plantilla actual, para pasar a la siguiente plantilla de relación o a la anterior, y para realizar otras funciones comunes.

Ahora veremos algunos ejemplos para ilustrar los recursos básicos de QBE. Debemos introducir explícitamente los operadores de comparación distintos de = (como > o ≥) antes de teclear un valor constante. Por ejemplo, la consulta C0A, "listar los números de seguro social de los empleados que trabajan más de 20 horas por semana en el proyecto número 1", se puede especificar como en la figura 8.4(a). Si las condiciones son más complejas, el usuario puede solicitar un **cuadro de condición**, que se crea pulsando una determinada tecla de función. En dicho cuadro, el usuario puede teclear la condición compleja.† Por ejemplo la consulta C0B, "listar los números de seguro social de los empleados que trabajan más de 20 horas por semana en el proyecto 1 o en el proyecto 2", se puede especificar como se muestra en la figura 8.4(b).

(a)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	José	B	Silva	_123456789	P._9/1/60	P._Norte 100, Higueras, MX	_M	_25000	_123456789	_3

(b)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	José	B	Silva		P._9/1/60	P._Norte 100, Higueras, MX				

(c)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	José	B	Silva		P._X	P._Y				

(d)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	José	B	Silva		P.	P.				

Figura 8.3 Cuatro formas de especificar la consulta C0 en QBE.

†La negación con el símbolo ¬ *no* es aceptable en un cuadro de condición.

Es posible especificar algunas condiciones complejas sin cuadros de condición. La regla es que todas las condiciones especificadas en la misma fila de una plantilla estén conectadas por la conectiva lógica **and** (para que una tupla sea seleccionada, deberá satisfacerlas *todas*), en tanto que las condiciones especificadas en distintas filas estén conectadas por **or** (*por lo menos una* debe satisfacerse). Por tanto, C0B también puede especificarse introduciendo dos filas distintas en la plantilla, como se muestra en la figura 8.4(c).

Consideremos ahora la consulta C0C, "listar los números de seguro social de los empleados que trabajan *tanto* en el proyecto 1 *como* en el proyecto 2". Esto no puede especificarse como en la figura 8.5(a), que lista aquellos que trabajan *ya sea* en el proyecto 1 *o* en el proyecto 2. La variable de ejemplo _ES se ligará a los valores de NSSE de las tuplas <–, 1, –> y *también* a los de las tuplas <–, 2, –>. La figura 8.5(b) muestra la forma de especificar C0C correctamente; la condición (_EX = _EY) del cuadro hará que las variables _EX y _EY se liguen sólo a valores idénticos de NSSE.

En general, una vez especificada una consulta, los valores resultantes se exhibirán en la plantilla dentro de las columnas apropiadas. Si el resultado contiene más filas de las que caben en la pantalla, la mayor parte de las implementaciones de QBE permitirán desplazarse verticalmente por las filas con la ayuda de una tecla de función. De manera similar, si una plantilla o varias son demasiado anchas para aparecer en la pantalla, se podrán desplazar horizontalmente para poder examinar todas las columnas.

Las operaciones de reunión se especifican en QBE colocando la *misma variable*† en las columnas que se van a reunir. Por ejemplo, la consulta C1, "listar el nombre y la dirección de todos los empleados que pertenecen al departamento 'Investigación'", se puede especificar como en la figura 8.6(a). Podemos especificar cualquier número de reuniones en una sola consulta, y también una **tabla de resultado** para exhibir el resultado de la consulta de reunión, como se ve en la figura 8.6(a); esto es necesario si el resultado incluye atributos de dos o más relaciones. Si no se especifica tabla de resultado, el sistema coloca el resultado de la consulta en las columnas de las diversas relaciones, lo que puede dificultar su interpretación.

(a)

TRABAJA_EN	NSSE	NÚMP	HORAS
	P.	1	>20

(b)

TRABAJA_EN	NSSE	NÚMP	HORAS
	P.	_PX	_HX

CONDICIONES

_HX>20 Y (_PX = 1 O _PX = 2)

(c)

TRABAJA_EN	NSSE	NÚMP	HORAS
	P.	1	>20
	P.	2	>20

Figura 8.4 Especificación de condiciones complejas en QBE. (a) La consulta C0A. (b) La consulta C0B con un cuadro de condición. (c) La consulta C0B sin cuadro de condición.

†Las variables se llaman **elementos de ejemplo** en los manuales de QBE.

(a)

TRABAJA_EN	NSSE	NÚMP	HORAS
	P._ES	1	
	P._ES	2	

(b)

TRABAJA_EN	NSSE	NÚMP	HORAS
	P._EX	1	
	P._EY	2	

CONDICIONES

_EX = _EY

Figura 8.5 Especificación de los empleados que trabajan en ambos proyectos. (a) Especificación incorrecta de una condición AND. (b) Especificación correcta.

La figura 8.6(a) ilustra también el mecanismo de QBE para especificar la obtención de todos los atributos de una relación, colocando el operador P. debajo del nombre de la relación en la plantilla de ésta.

Si queremos reunir una tabla consigo misma, especificaremos diferentes variables que representen las distintas referencias a la tabla. Por ejemplo, la consulta C8, "para cada empleado obtener su nombre y apellido, así como el nombre y el apellido de su supervisor inmediato", se puede especificar como en la figura 8.6(b), donde las variables que comienzan con E se refieren a un empleado y las que comienzan con S se refieren a un supervisor.

Consideremos ahora los tipos de consultas que requieren agrupación o funciones agregadas. Podemos especificar un operador de agrupación, G., en una columna para indicar que las tuplas deben agruparse según el valor de esa columna. Es posible especificar funciones comunes, como AVG. (promedio), SUM., CNT. (cuenta), MAX. y MIN. En QBE las funciones AVG., SUM. y CNT. se aplican a *valores distintos* dentro de un grupo en el caso

(a)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	_NP		_AP			_DIR				_DX

DEPARTAMENTO	NOMBRED	NÚMEROD	NSSGTE	FECHAINICGTE
	Investigación	_DX		

RESULTADO			
P.	_NP	_AP	_DIR

(b)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	_E1		_E2						_XNSS	
	_S1		_S2		_XNSS					

RESULTADO				
P.	_E1	_E2	_S1	_S2

Figura 8.6 Ilustración de la REUNIÓN y de las relaciones de resultado en QBE. (a) La consulta C1. (b) La consulta C8.

(a)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
								P.CNT.		

(b)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
								P.CNT.ALL		

(c)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
					P.CNT.ALL			P.AVG.ALL		P.G.

(d)

PROYECTO	NOMBREPR	NÚMEROP	LUGARP	NÚMD
	P.	_PX		

TRABAJA_EN	NSSE	NÚMP	HORAS
	P.CNT._EX	G._PX	

CONDICIONES

CNT._EX>2

Figura 8.7 Funciones y agrupación en QBE. (a) La consulta C19. (b) La consulta C19A. (c) La consulta C20. (d) La consulta C22A.

por omisión. Si queremos que estas funciones se apliquen a todos los valores, deberemos usar el prefijo ALL.† Esta convención es *diferente* en SQL y QUEL, donde por omisión se aplicará una función a todos los valores.

La figura 8.7(a) muestra la consulta C19, que cuenta el número de valores de salario *distintos* en la relación EMPLEADO. La consulta C19A (Fig. 8.7(b)) cuenta todos los valores de salario, lo que equivale a contar el número de empleados. La figura 8.7(c) muestra C20, que obtiene el número de cada departamento, el número de empleados que pertenecen a él y el salario medio dentro de cada departamento; por tanto, la columna ND sirve para agrupar, como lo indica la función G. Podemos especificar varios de los operadores G., P. y ALL en una sola columna. La figura 8.7(d) muestra la consulta C22A, que exhibe el nombre de cada proyecto en el que trabajan más de dos empleados y el número de empleados que trabajan en él.

QBE tiene un símbolo de negación, ¬, que se utiliza de manera similar a la función NOT EXISTS de SQL. La figura 8.8 muestra la consulta C6, que lista los nombres de los empleados que no tienen dependientes. El símbolo de negación indica que se seleccionarán valores de la variable _SX de la relación EMPLEADO sólo si no ocurren en la relación DEPENDIENTE. El mismo efecto podría lograrse escribiendo ≠ _SX en la columna NSSE.

Aunque la propuesta original del lenguaje QBE manejaba el equivalente de las funciones EXISTS y NOT EXISTS de SQL, la implementación de QBE en QMF (bajo el sistema DB2) *no* lo hace. Por tanto, la versión QMF de QBE que hemos analizado aquí *no es relacionalmente completa*. Las consultas como C3, "buscar los empleados que trabajan en *todos* los proyectos controlados por el departamento 5", *no se pueden* especificar.

QBE tiene tres operadores para actualizar la base de datos: I. para insertar, D. para eliminar y U. para modificar. Los operadores de inserción y eliminación se especifican en la columna de la plantilla bajo el nombre de la relación, pero el operador de modificación se

†ALL de QBE no tiene que ver con el cuantificador universal. Asimismo, ALL se utiliza en QUEL con un propósito distinto.

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	P.		P.	_SX						

DEPENDIENTE	NSSE	NOMBRE_DEPENDIENTE	SEXO	FECHAN	PARENTESCO
¬	_SX				

Figura 8.8 Ilustración de la negación mediante la consulta C6.

especifica en las columnas que se van a modificar. La figura 8.9(a) muestra la forma de insertar una nueva tupla EMPLEADO. Para la eliminación, primero introducimos el operador D. y luego especificamos las tuplas que se eliminarán mediante una condición (Fig. 8.9(b)). Para modificar una tupla, especificamos el operador U. bajo el nombre del atributo, seguido del nuevo valor del atributo. También deberemos seleccionar de la manera acostumbrada la tupla o tuplas que se van a modificar. La figura 8.9(c) muestra una solicitud de modificación para aumentar el salario de 'José Silva' en un 10% y además para reasignarlo al departamento 4.

QBE también cuenta con mecanismos para definir los datos. Las tablas de una base de datos se pueden especificar interactivamente, y las definiciones de tablas también pueden actualizarse añadiendo o eliminando una columna, o cambiando su nombre. Asimismo, podemos especificar diversas características para cada columna: si es la clave de la relación, el tipo de datos que tiene, si debe crearse un índice sobre ese campo, etc. QBE cuenta además con mecanismos para definir vistas, autorizar el acceso, almacenar definiciones de consultas para su uso posterior, y así sucesivamente.

QBE no tiene un estilo "lineal" como QUEL y SQL; más bien, es un lenguaje "bidimensional", porque los usuarios especifican sus consultas desplazándose por toda el área de la pantalla. Pruebas con usuarios han demostrado que QBE es más fácil de aprender que SQL, sobre todo para quienes no son especialistas. En este sentido, QBE fue el primer lenguaje de bases de datos relacionales amable con el usuario.

En fechas más recientes se han creado muchas otras interfaces amables con el usuario para los sistemas comerciales de bases de datos. El empleo de menús, gráficos y formas es ahora cosa muy común.

8.5 Resumen

El cálculo relacional es la expresión formal de un lenguaje de consulta declarativo para el modelo relacional, y se basa en la rama de la lógica matemática llamada cálculo de predicados.

(a)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
I.	Ricardo	C	Martínez	653298653	30-DIC-52	Olmo 98, Cedros, MX	M	37000	987654321	4

(b)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
D.				653298653						

(c)

EMPLEADO	NOMBREP	INIC	APELLIDO	NSS	FECHAN	DIRECCIÓN	SEXO	SALARIO	NSSSUPER	ND
	José		Silva					U._S*1.1		U.4

Figura 8.9 Actualización de la base de datos en QBE. (a) Inserción. (b) Eliminación. (c) Modificación en QBE.

Hay dos tipos de cálculos relacionales: el cálculo relacional de tuplas emplea variables que abarcan relaciones; el cálculo relacional de dominios utiliza variables de dominio.

Una consulta se especifica en un solo enunciado declarativo, sin especificar ningún orden ni método para obtener el resultado de la consulta. Por esto, muchos consideran que el cálculo relacional es un lenguaje de más alto nivel que el álgebra relacional. Las expresiones del álgebra relacional especifican implícitamente un ordenamiento de las operaciones para obtener el resultado de una consulta, en tanto que las expresiones del cálculo relacional sólo especifican lo que se desea obtener, independientemente de cómo se pueda ejecutar la consulta.

Estudiamos la sintaxis de las consultas en el cálculo relacional, así como los cuantificadores existencial (∃) y universal (∀). Vimos que las variables del cálculo relacional están ligadas a estos cuantificadores y examinamos con detalle la forma de escribir consultas con cuantificación universal, analizando el problema de especificar consultas seguras cuyos resultados sean finitos. También presentamos reglas para transformar los cuantificadores universales en existenciales, y viceversa. Son los cuantificadores los que confieren poder de expresión al cálculo relacional, haciéndolo equivalente al álgebra relacional.

El lenguaje SQL, que describimos en el capítulo 7, tiene sus raíces en el cálculo relacional de tuplas. Una consulta de SELECCIONAR–PROYECTAR–REUNIR en SQL es similar a una expresión del cálculo relacional de tuplas, si consideramos cada nombre de relación especificado en la cláusula FROM de la consulta SQL como una variable de tupla con un cuantificador existencial implícito. La función EXISTS de SQL equivale al cuantificador existencial y se puede usar en su forma negada (NOT EXISTS) para especificar cuantificación universal. SQL no cuenta con un equivalente explícito del cuantificador universal. La agrupación y las funciones agregadas no tienen contraparte en el cálculo relacional.

También analizamos dos lenguajes de bases de datos que tienen que ver con el cálculo relacional. El lenguaje QUEL es muy parecido al cálculo relacional de tuplas, sin el cuantificador universal. El lenguaje QBE tiene similitudes con el cálculo relacional de dominios.

Preguntas de repaso

8.1. ¿En qué sentido difiere el cálculo relacional del álgebra relacional, y en qué sentido es similar?

8.2. ¿Qué diferencias hay entre el cálculo relacional de tuplas y el cálculo relacional de dominios?

8.3. Explique los significados del cuantificador existencial (∃) y del cuantificador universal (∀).

8.4. Defina los siguientes términos con respecto al cálculo de tuplas: *variable de tupla, relación de intervalo, átomo, fórmula, expresión.*

8.5. Defina los siguientes términos con respecto al cálculo de dominios: *variable de dominio, relación de intervalo, átomo, fórmula, expresión.*

8.6. ¿Qué queremos decir con *expresión segura* en el cálculo relacional?

8.7. ¿Cuándo se dice que un lenguaje de consulta es relacionalmente completo?

8.8. Analice las reglas para especificar atributos de agrupación y funciones en QUEL.

8.9. Analice las reglas para anidar funciones en QUEL.

8.10. Compare las diversas características de SQL y QUEL, y explique por qué podría preferir uno al otro según cada una de sus características.

8.11. Analice las reglas para anidar operadores en QBE.

8.12. ¿Por qué deben aparecer los operadores de inserción (I.) y eliminación (D.) de QBE abajo del nombre de la relación en una plantilla, y no debajo del nombre de una columna?

8.13. ¿Por qué deben aparecer los operadores de modificación (U.) debajo de un nombre de columna en una plantilla de relación y no abajo del nombre de la relación?

Ejercicios

8.14. Especifique las consultas a, b, c, e, f, i y j del ejercicio 6.19 tanto en el cálculo relacional de tuplas como en el cálculo relacional de dominios.

8.15. Especifique las consultas a, b, c y d del ejercicio 6.21 tanto en el cálculo relacional de tuplas como en el de dominios.

8.16. Especifique las consultas del ejercicio 7.16 tanto en el cálculo relacional de tuplas como en el de dominios. Además, especifique esas consultas en el álgebra relacional.

8.17. En una consulta del cálculo relacional de tuplas con n variables de tupla, ¿cuál sería el número mínimo característico de condiciones de reunión? ¿Por qué? ¿Qué sucedería si tuviéramos menos condiciones de reunión?

8.18. Reescriba las consultas del cálculo relacional de dominios que siguieron a C0 en la sección 8.3 utilizando la notación abreviada de C0A, donde el objetivo es minimizar el número de variables de dominio escribiendo constantes en lugar de variables siempre que esto sea posible.

8.19. Considere esta consulta: obtener los NSS de los empleados que trabajan en por lo menos aquellos proyectos en los cuales trabaja el empleado con NSS = 123456789. Esto puede expresarse como (PARA TODO x) (SI P ENTONCES Q), donde:

- x es una variable de tupla que abarca la relación PROYECTO.
- $P \equiv$ empleado con NSS = 123456789 que trabaja en el proyecto x.
- $Q \equiv$ empleado e que trabaja en el proyecto x.

Exprese la consulta en el cálculo relacional de tuplas, con base en las reglas:

- $(\forall x)(P(x)) \equiv (\nexists x)(\text{not}(P(x)))$.
- (SI P ENTONCES Q) $\equiv$ (not(P) or Q).

8.20. Indique la forma de especificar las siguientes operaciones del álgebra relacional tanto en el cálculo relacional de tuplas como en el de dominios.

a. $\sigma_{A=C}(R(A, B, C))$.

b. $\pi_{<A,B>}(R(A, B, C))$.

c. $R(A, B, C) * S(C, D, E)$.

d. $R(A, B, C) \cup S(A, B, C)$.

e. $R(A, B, C) \cap S(A, B, C)$.

f. $R(A, B, C) - S(A, B, C)$.

g. $R(A, B, C) \times S(D, E, F)$.

h. $R(A, B) \div S(A)$.

8.21. Sugiera extensiones del cálculo relacional para que sea posible expresar los siguientes tipos de operaciones que vimos en la sección 6.6: (a) funciones agregadas y agrupación; (b) operaciones de REUNIÓN EXTERNA; (c) consultas de cerradura recursiva.

8.22. Escriba enunciados de definición de datos apropiados en QUEL para algunos de los esquemas de base de datos que se muestran en las figuras 6.5, 2.1, 6.19 y 6.22.

8.23. Especifique las consultas de los ejercicios 6.19 y 7.14 en QUEL.

8.24. Considere la siguiente consulta, que es una variación de C24: obtener el NSS del individuo más joven del departamento 5 entre los empleados que ganan más de $50 000. ¿Se puede hacer esto con una sola consulta de QUEL? ¿Se puede hacer en SQL? Si no puede efectuarse con una sola consulta, muestre cómo podría hacerse en pasos almacenando resultados temporales.

8.25. Especifique las actualizaciones del ejercicio 6.20 empleando las órdenes de actualización de QUEL.

8.26. Especifique las consultas del ejercicio 7.16 en QUEL.

8.27. Especifique las actualizaciones del ejercicio 7.17 empleando las órdenes de actualización de QUEL.

8.28. Especifique las consultas del ejercicio 6.26 en QUEL.

8.29. Escriba enunciados QUEL para crear los índices especificados en el ejercicio 7.19.

8.30. Especifique las consultas y actualizaciones de los ejercicios 6.21 y 6.22 en QUEL.

8.31. Repita el ejercicio 7.25, pero use QUEL en vez de SQL.

8.32. Especifique algunas de las consultas de los ejercicios 6.19 y 7.14 en QBE.

8.33. ¿Puede especificar la consulta del ejercicio 8.24 como una sola consulta en QBE?

8.34. Especifique las actualizaciones del ejercicio 6.20 en QBE.

8.35. Especifique las consultas del ejercicio 7.16 en QBE.

8.36. Especifique las actualizaciones del ejercicio 7.17 en QBE.

8.37. Especifique las consultas y actualizaciones de los ejercicios 6.21 y 6.22 en QBE.

Bibliografía selecta

Codd (1971) presentó el lenguaje ALPHA, que se basa en conceptos del cálculo relacional de tuplas. ALPHA incluye también la noción de función agregada, lo que va más allá del cálculo relacional. La primera definición formal del cálculo relacional aparece en Codd (1972), donde también se presenta un algoritmo para transformar cualquier expresión del cálculo relacional de tuplas al álgebra relacional. Según la definición de Codd, un lenguaje es relacionalmente completo si es por lo menos tan potente como el cálculo relacional. Ullman (1988) describe una demostración formal de la equivalencia entre el álgebra relacional y las expresiones seguras de los cálculos relacionales de tuplas y de dominios.

Las ideas del cálculo relacional de dominios aparecieron por primera vez en el lenguaje QBE (Zloof 1975). Lacroix y Pirotte (1977) publicaron la definición formal de este concepto. El lenguaje ILL (Lacroix and Pirotte 1977*a*) se basa en el cálculo relacional de dominios. El lenguaje QUEL (Stonebraker *et al.* 1976) se basa en el cálculo relacional de tuplas, con cuantificadores existenciales implícitos pero sin cuantificadores universales, y se implementó en el sistema INGRES. Zook *et al.* (1977) describe el lenguaje para el "INGRES universitario", y la versión comercial de QUEL se describe en RTI (1983). Un libro (Stonebraker 1986) contiene una compilación de artículos de investigación y síntesis relacionados con el sistema INGRES.

Thomas y Gould (1975) informan los resultados de experimentos que comparan la facilidad de uso de QBE con la de SQL. Las funciones del QBE comercial se describen en un manual de IBM (1978); existe una tarjeta de referencia rápida (IBM 1978*a*), y en los manuales de referencia de DB2 apropiados se examina la implementación de QBE para ese sistema. Whang *et al.* (1990) integra los cuantificadores universales a las capacidades de QBE.

CAPÍTULO 9

Un sistema de gestión de bases de datos relacionales: DB2

En este capítulo analizaremos un ejemplo representativo de un sistema de gestión de bases de datos relacionales (SGBDR): DB2. La arquitectura básica de DB2 se bosqueja en la sección 9.2. En la sección 9.3 se explica la definición de datos en DB2 y en la sección 9.4 se ve cómo se manipulan los datos. La sección 9.5 es un panorama de las estructuras de almacenamiento de DB2, y en la sección 9.6 se analizan algunas características adicionales. En primer lugar, en la sección 9.1, presentaremos una perspectiva histórica sobre el desarrollo de los sistemas de bases de datos relacionales.

Al lector que sólo requiera una introducción general al sistema DB2 puede pasar por alto las secciones 9.5 y 9.6, que tratan aspectos físicos de DB2.

9.1 Introducción a los sistemas de gestión de bases de datos relacionales

Después de que Codd presentó el modelo relacional en 1970, muchos investigadores se apresuraron a experimentar con las ideas relacionales. Un proyecto importante de investigación y desarrollo se inició en el San Jose (ahora llamado Almaden) Research Center de IBM. Éste condujo a la puesta en el mercado de dos productos de SGBD relacionales de IBM en la década de 1980: SQL/DS, para los entornos DOS/VSE (*disk operating system/virtual storage extended:* sistema operativo de disco/almacenamiento virtual extendido), y VM/CMS (*virtual machine/conversational monitoring system:* máquina virtual/sistema de vigilancia conversacional), presentados en 1981; y DB2 para el sistema operativo MVS, presentado en 1983. Otro importante SGBD relacional es INGRES, creado en la University of California, Berkeley, a principios de la década de 1970 y comercializado por Relational Technology Inc., a finales de esa misma década. Ahora INGRES es un SGBDR comercial vendido por Ingres, Inc., una subsidiaria de ASK, Inc. Hay otros SGBDR en el mercado que se han popularizado: ORACLE de Oracle, Inc.; Sybase de Sybase, Inc.; RDB de Digital Equipment Corp.; INFORMIX de Informix, Inc., y UNIFY de Unify, Inc. No es posible describir con detalle las características de cada uno de estos SGBDR. En este capítulo estudiaremos las características del producto DB2 de IBM para que el lector tenga una idea de lo que suelen ofrecer los productos de SGBDR comerciales. En los demás sistemas, las arquitecturas, el apoyo de lenguajes y las herramientas para crear aplicaciones son similares.

Además de los SGBDR antes mencionados, en la década de 1980 muchas implementaciones del modelo relacional de los datos aparecieron en la plataforma del computador personal (PC). Entre ellas están RIM, RBASE 5000, PARADOX, OS/2 Database Manager, DBase IV, XDB, WATCOM SQL, SQL Server (de Sybase, Inc.) y, en fechas más recientes ACCESS (de Microsoft, Inc.). En un principio, éstos fueron sistemas para un solo usuario, pero últimamente han comenzado a ofrecer la arquitectura de bases de datos cliente/servidor (véase la definición en el Cap. 23) y se han venido ajustando a la norma *Open Database Connectivity* (ODBC: conectividad abierta de bases de datos) de Microsoft. Gracias a esta norma nos es posible emplear muchas herramientas para máquina frontal con estos sistemas. No examinaremos los SGBDR para PC aquí.

Además, algunos proveedores utilizan de manera un tanto inapropiada el término *relacional*, con fines publicitarios, al hablar de sus productos. Para calificar como SGBD relacional genuino, un sistema debe poseer por lo menos las siguientes propiedades:[†]

1. Debe almacenar los datos como relaciones de modo que cada columna se identifique de manera independiente por su nombre de columna y el ordenamiento de las filas carezca de importancia.
2. Las operaciones que puede realizar el usuario, así como las que utiliza internamente el sistema, deben ser verdaderas operaciones relacionales; esto es, deben ser capaces de generar nuevas relaciones a partir de otras ya existentes.
3. El sistema debe manejar por lo menos una variante de la operación REUNIÓN.

Aunque podríamos hacer más larga esta lista, proponemos estos criterios como un conjunto mínimo para determinar si un sistema es o no relacional. Es fácil comprobar que muchos SGBD que se promueven como relacionales no satisfacen estos criterios.

A continuación describiremos DB2, que es uno de los sistemas relacionales para computador central más utilizados en la actualidad.

9.2 Arquitectura básica de DB2

El nombre DB2 es una abreviatura de Database 2. Es un producto SGBD relacional de IBM para el sistema operativo MVS. El presente análisis se refiere a DB2 Version 2 Release 3 (segunda versión, tercera edición) y, hasta cierto punto, a DB2 Version 3. Haremos hincapié en comunicar la complejidad de un producto de SGBDR como DB2 y su gran variedad de funciones,

[†]Codd (1985) especificó 12 reglas para determinar si un SGBD es relacional, y las presentamos en un apéndice al final del capítulo. Codd (1990) presenta un tratado sobre los modelos y sistemas relacionales extendidos, identificando más de 330 características de los sistemas relacionales, divididas en 18 categorías.

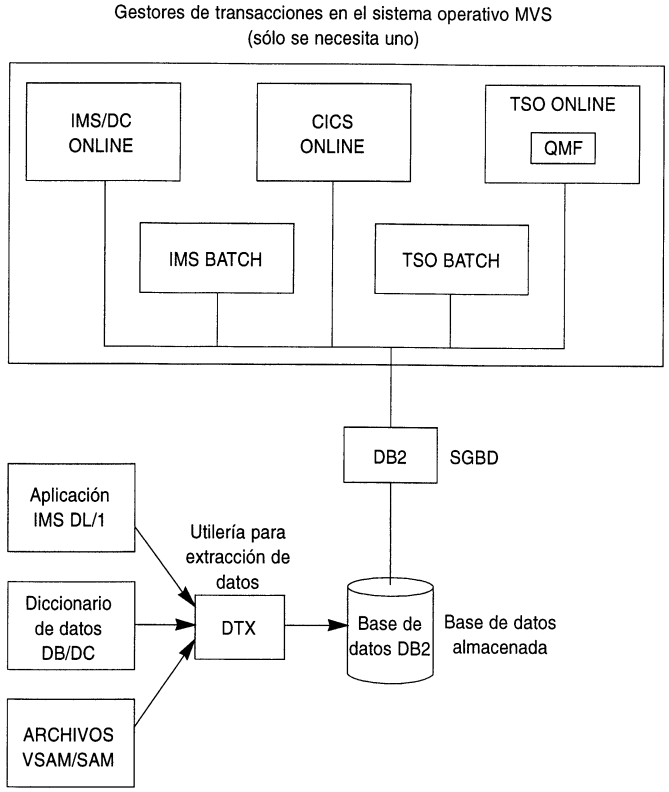


Figura 9.1 Panorama de la organización del sistema DB2, con una vista parcial de sus características.

en vez de presentar una enumeración exacta de las características y los recursos de una versión específica de DB2.

DB2 coopera con ("se anexa a", en la terminología del producto) cualquiera de los tres entornos de subsistema de MVS: CICS, TSO e IMS. Estos sistemas cooperan con los recursos de DB2 para suministrar comunicación de datos y gestión de transacciones. La figura 9.1 muestra las relaciones entre los diversos componentes de DB2. DB2 proporciona acceso concurrente a las bases de datos para los usuarios de IMS/VS-DC (*Information Management System/Virtual Storage-Data Communications:* sistema de gestión de información/almacenamiento virtual-comunicación de datos), CICS (*Customer Information Control System:* sistema de control de información de clientes) y TSO (*Time Sharing Option:* opción de tiempo compartido), tanto interactivos como por lotes. CICS es un sistema de supervisión de teleproceso: un producto de IBM que muchas industrias utilizan para procesar transacciones de negocios. (Hablaremos del procesamiento de transacciones en el capítulo 17.) IMS/DC es un entorno de comunicación de datos que maneja bases de datos IMS jerárquicas (analizaremos IMS en la sección 11.7). TSO es un entorno de tiempo compartido producido por IBM y

en uso desde hace muchos años. Con el recurso "llamar y anexar" CAF (*Call Attach Facility*; no se muestra en la figura 9.1) una aplicación puede interactuar con una base de datos DB2 sin la ayuda de estos supervisores. Las bases de datos DB2 se pueden utilizar desde programas de aplicación escritos en COBOL, PL/1, FORTRAN, C, PROLOG o lenguaje ensamblador de IBM.

Los puntos que siguen describen con mayor detalle el empleo de los diversos subsistemas ilustrados en la figura 9.1:

1. Una aplicación DB2 consistente en programas escritos en los lenguajes antes mencionados se ejecuta bajo el control de *uno y sólo uno* de los tres subsistemas: IMS, CICS o TSO. Las aplicaciones DB2 IMS, CICS y TSO se ejecutan por separado; de hecho, se describen en distintos juegos de manuales.
2. Las aplicaciones IMS, CICS y TSO pueden compartir las mismas bases de datos DB2. CSP (*Cross Systems Product:* producto intersistemas) hace posible ejecutar bajo CICS una aplicación creada bajo TSO, y viceversa.
3. DB2 ofrece dos recursos en línea principales: QMF (*Query Management Facility:* recurso de gestión de consultas) bajo CICS o CAF, y DB2 Interactive (DB2I) bajo TSO. DB2I acompaña a DB2; permite a los usuarios profesionales introducir SQL interactivamente a través de una interfaz llamada SPUFI y les ayuda a preparar programas y utilerías para su ejecución.
4. Además de las bases de datos DB2, las bases de datos IMS también son accesibles desde una aplicación DB2 bajo los entornos IMS o CICS, pero no bajo TSO. La misma aplicación TSO se puede ejecutar por lotes o en línea, dirigiendo la E/S del programa a archivos o usando terminales para E/S.

Como se muestra en la figura 9.1, otros dos recursos —QMF y DXT— desempeñan importantes papeles en el empleo de DB2.

QMF (Query Management Facility). QMF es un producto de IBM que se vende por separado y que actúa como lenguaje de consulta y elaborador de informes. Se ejecuta simplemente como aplicación TSO en línea. Permite a usuarios finales no técnicos hacer consultas *ad hoc* en SQL o en QBE y muestra los resultados de dichas consultas como informes con formato. Puede tener acceso a bases de datos DB2 y SQL/DS. La salida de QMF se puede dirigir a otras utilerías a fin de dibujar gráficas de barras (con la utilería interactiva de graficación, *Interactive Chart Utility*) y otras presentaciones gráficas de los datos (con el administrador de presentación gráfica de datos, *Graphical Data Display Manager*). Los usuarios construyen formas de manera interactiva y así controlan la presentación de los resultados de la consulta.

DXT (Data Extract). Éste es un programa de utilería que extrae datos de bases de datos IMS o de archivos VSAM (*Virtual Storage Access Method:* método de acceso de almacenamiento virtual) o SAM (*Sequential Access Method:* método de acceso secuencial) y los convierte en un archivo secuencial. DXT puede especificar las solicitudes de extracción en forma interactiva o como trabajos por lotes. Este archivo secuencial tiene un formato adecuado para cargarse directamente en una base de datos DB2. DXT es un producto independiente de IBM.

9.2.1 *La familia DB2 y DB2/2*

SQL/DS es el primer SGBD relacional de IBM que pertenece a la "familia DB2". Los recursos de definición y manipulación de datos en ambos sistemas son esencialmente idénticos, y sólo muestran diferencias sintácticas secundarias. Los dos emplean SQL como lenguaje de consulta interactiva y como lenguaje de programación de bases de datos, pues lo incorporan en un lenguaje anfitrión (véase la Sec. 7.7). En un principio, DB2 usaba el código SQL/DS para las partes superiores del sistema (por ejemplo, las funciones de optimización y procesamiento de consultas). Las partes "inferiores" de DB2 se construían a partir de cero. Los recursos QMF y DXT pueden usarse tanto con DB2 como con SQL/DS; sin embargo, el almacenamiento de datos en SQL/DS y DB2 es diferente. Por ejemplo, conceptos como los espacios de tabla de DB2 no tienen contraparte en SQL/DS. Además, DB2 permite usar técnicas especializadas para manejar bases de datos grandes y cargas de trabajo pesadas que son exclusivas del sistema operativo MVS. Así como DB2 proporciona un recurso SQL interactivo para los usuarios finales a través de DB2I, SQL/DS puede ofrecer a éstos algunos recursos a través de su componente ISQL (SQL interactivo). El IBM Database 2 OS/2 (abreviado DB2/2) es el último de los SGBDR de IBM. Se ejecuta bajo OS/2 y se introdujo en 1988 con el nombre OS/2 Extended Edition Database Manager. Cabe señalar que el SGBD AS/400 DBMS es diferente de los SGBDR de la familia DB2. La mayor parte de su funcionalidad reside en hardware propietario.

9.2.2 *Organización de datos y procesos en DB2*

Veamos primero cómo los usuarios y los programas de aplicación perciben las bases de datos DB2. Todos los datos se visualizan como relaciones o "tablas" (término legítimo en DB2). Las tablas son de dos tipos: *tablas base*, que existen físicamente como datos almacenados, y *vistas*, que son tablas virtuales sin identidad física independiente en el almacenamiento. Una tabla base consiste en uno o más archivos VSAM.

Procesamiento de aplicaciones. En la figura 9.2 se muestra de manera simplificada la preparación de una aplicación DB2 en SQL incorporado. Indica la secuencia de procesos por la que deben pasar las aplicaciones de los usuarios para tener acceso a una base de datos DB2. Los componentes principales del flujo de la aplicación SQL son el precompilador, el enlazador (*Bind*), el supervisor durante la ejecución† y el manejador de datos almacenados. En pocas palabras, realizan las siguientes funciones:

- **Precompilador:** Como se explicó en los capítulos 1 y 7, la tarea del precompilador es procesar las instrucciones de SQL incorporadas en un programa escrito en un lenguaje anfitrión. El precompilador genera dos tipos de salidas: el programa fuente original, donde las instrucciones de SQL incorporado son reemplazadas por llamadas CALL; y módulos de solicitud a la base de datos (DBRM: *database request module*), que son colecciones de instrucciones de SQL en forma de árbol de análisis sintáctico y que constituyen la entrada al proceso de enlace.

†Los términos *supervisor durante la ejecución* (*run-time supervisor*) y *manejador de datos almacenados* (*stored data manager*) no aparecen en los manuales de DB2. Los hemos adaptado de Date y White (1988). El término *Relational Data System* (RDS: sistema de datos relacional) que aparece en los manuales de IBM proviene de la implementación original de System R; abarca el precompilador, el enlazador y el supervisor durante la ejecución.

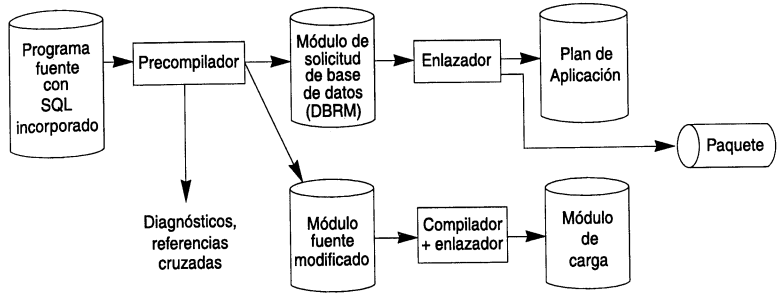


Figura 9.2 Preparación de aplicaciones.

- **Enlazador:** Este componente atiende ambos tipos de solicitudes de SQL: aquellas que aparecen en programas de aplicación que han de ejecutarse una y otra vez, y las consultas *ad hoc* que se ejecutan sólo una vez. En la primera categoría, después de un análisis detallado y una optimización de consulta (como se explicará en el capítulo 16), uno o más DBRM relacionados se compilan *sólo una vez* para producir un plan de aplicación. Así, como el costo de este enlace se amortiza durante las múltiples ejecuciones del programa, se justifica plenamente. Además, el proceso de enlace permite a una aplicación construir e introducir una instrucción en SQL (dinámico) para que se ejecute de inmediato. El enlazador analiza sintácticamente todos los enunciados SQL. Las instrucciones de manipulación de SQL se enlazan para producir código ejecutable, pero los enunciados de definición y control sólo se analizan y se dejan en una forma que se interpreta en el momento de la ejecución. La salida del enlazador se denomina plan o paquete.

- **Supervisor durante la ejecución:** Con este término nos referimos a los servicios de DB2 que controlan la ejecución de la aplicación. La ejecución de una llamada SQL dentro de un programa de aplicación real sigue los pasos de procesamiento ilustrados en la figura 9.3. Cuando se está ejecutando el módulo de carga del programa de aplicación y llega a una instrucción CALL insertada por el precompilador, el control pasa al supervisor a través del módulo apropiado de interfaz de lenguaje de DB2. El supervisor durante la ejecución obtiene el plan de aplicación y, aprovechando la información de control en éste, solicita que el manejador de datos almacenados tenga acceso real a la base de datos.

- **Manejador de datos almacenados:** Éste es el componente del sistema que maneja la base de datos física. Contiene lo que en DB2 se llama Data Manager (gestor de datos), así como el Buffer Manager (gestor de almacenamiento intermedio), Log Manager (gestor de bitácora), etc. Juntos realizan todas las funciones necesarias para manipular la base de datos almacenada —buscar, leer, actualizar— según los requerimientos del plan de aplicación. Este componente actualiza los índices si es necesario. A fin de lograr el desempeño óptimo de las reservas de almacenamiento

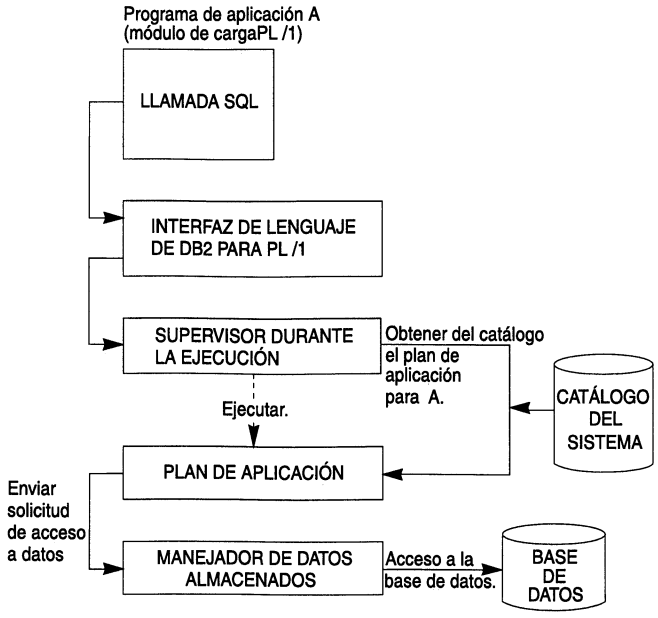


Figura 9.3 Ejemplo de ejecución de una aplicación PL/1 en DB2.

intermedio de lectura anticipada (*read-ahead*) y el de buscar al lado (*look-aside*). El manejador de datos almacenados puede otorgar acceso disperso y enlazado a las tablas del sistema almacenadas en el catálogo; tiene acceso a los datos o a los índices proporcionando identificadores de páginas al gestor de almacenamiento intermedio. El tamaño de las páginas es de 4096 bytes y corresponde al tamaño de página del sistema operativo.

Procesamiento interactivo. Los usuarios en línea de DB2 pueden tener acceso a la base de datos mediante el recurso DB2I (DB2 interactivo), que es una aplicación en línea ejecutable bajo DB2. Para que un usuario en línea pueda comunicarse con esta aplicación en línea requiere los servicios de un gestor de comunicación de datos (DC: *data communication*). En el caso de DB2, la función de gestor de comunicación de datos la desempeña el componente TSO de MVS, el recurso de comunicación de datos DC de IMS, o el CICS (véase la Fig. 9.1). DB2I acepta instrucciones en SQL de una terminal y los transfiere a DB2 para que los ejecute. Incluso durante la ejecución interactiva, las instrucciones de SQL se compilan y se genera el plan de aplicación correspondiente; los resultados se devuelven a la terminal y el plan se desecha después de la ejecución.

Utilerías. Los servicios de base de datos incluyen un conjunto de utilerías, las cuales describiremos más adelante (Sec. 9.5.3).

9.2.3 *Otras funciones relacionadas con la compilación y ejecución de SQL*

Se requieren varias funciones durante la compilación y ejecución de consultas o aplicaciones SQL. Aquí sólo daremos una rápida explicación de las más importantes:

- *Optimización:* Esta función se encarga de elegir un plan de acceso óptimo para atender una solicitud de obtención o actualización en SQL.
- *Recompilación de planes de aplicación:* Siempre que se crea o desecha un índice con CREATE INDEX o DROP INDEX, el supervisor durante la ejecución marca como "no válido" el plan de aplicación correspondiente en el catálogo. Si hay una invocación subsecuente de un plan no válido, se llama al enlazador para que lo recompile con base en el conjunto vigente de índices. Todo el proceso de *enlace automático* es transparente para el usuario.
- *Comprobación de autorización:* El enlazador también verifica si el usuario que lo invocó está autorizado para llevar a cabo las operaciones incluidas en los DBRM que se van a enlazar. DB2 usa un identificador de autorización del solicitante para determinar si tiene privilegios de acceso. IMS y CICS proveen los identificadores de autorización y controlan su empleo. TSO emplea por omisión el identificador del usuario. Cada uno de los entornos de conexión informa a DB2, con el recurso IDENTIFY, cuál es su tipo de conexión. A fin de evitar solicitudes IDENTIFY no autorizadas, una instalación puede controlar quién tiene permiso para conectarse con DB2 y qué tipo de conexiones puede usar.

9.3 Definición de datos en DB2

Ya estudiamos en la sección 7.1 qué recursos de definición de datos tiene SQL. Estos recursos permiten crear, eliminar y alterar (cuando sea apropiado) tablas base, vistas e índices. Entre sus instrucciones se encuentran las siguientes:

Para tablas	Para vistas	Para índices
CREATE TABLE	CREATE VIEW	CREATE INDEX
ALTER TABLE		ALTER INDEX
DROP TABLE	DROP VIEW	DROP INDEX

No existe la instrucción ALTER VIEW (alterar vista). ALTER INDEX (alterar índice) sí existe, pero maneja los parámetros físicos de los índices. En el capítulo 7 (Fig. 7.1) presentamos una definición completa de una base de datos relacional en términos de instrucciones CREATE TABLE (crear tabla). Durante la creación, no se impone explícitamente a las tablas base ningún ordenamiento de las tuplas. El ordenamiento de las columnas es implícito en virtud del orden en que están los nombres de las columnas en la instrucción CREATE TABLE.

DB2 maneja el concepto de valores nulos. Cualquier columna puede contener un valor nulo a menos que la definición de esa columna en CREATE TABLE especifique explícitamente NOT NULL. Las columnas en las que se permiten valores nulos se representan físicamente en la base de datos almacenada mediante dos columnas: la columna de datos propiamente dicha y una columna oculta de indicadores, con un byte de ancho, en la que un valor hexadecimal de FF significa que se debe ignorar el valor de datos correspondiente (es decir, es nulo) y un valor de 00 indica que el valor correspondiente es válido (no nulo).

9.3.1 *El catálogo del sistema*

En DB2, el administrador de bases de datos u otros usuarios autorizados pueden tener acceso, a través de SQL, a las tablas denominadas *catálogo*. El catálogo del sistema DB2 contiene una gran variedad de información, como definiciones de tablas base, vistas, índices, aplicaciones, usuarios, privilegios de acceso y planes de aplicación. El sistema consulta estas descripciones para efectuar ciertas tareas; por ejemplo, durante la optimización de consultas el componente enlazador usa el catálogo para obtener información sobre los índices.

DB2 adopta un enfoque uniforme para almacenar los datos y el catálogo: ambos se almacenan como tablas. En vez de describir exhaustivamente el catálogo, daremos una idea de su contenido mencionando unas cuantas tablas importantes:

1. SYSTABLES: Ésta tiene una entrada por cada tabla base del sistema. La información relativa a cada tabla incluye, entre otras cosas, su nombre, el nombre de quien la creó y el número total de columnas que contiene.
2. SYSCOLUMNS: Ésta contiene una entrada por cada columna (atributo) definida en el sistema. Para cada columna, se almacena su nombre, el nombre de la tabla a la que pertenece, su tipo e información adicional. El mismo nombre de columna puede aparecer en varias tablas.
3. SYSINDEXES. Ésta contiene, para cada índice, su nombre, el nombre de la tabla indizada, el nombre del usuario que creó el índice, etcétera.

Consulta de la información del catálogo. Como el catálogo está organizado en forma de tablas, puede consultarse con SQL, igual que cualquier otra tabla. Por ejemplo, consideremos la consulta

```
SELECT  NAME
FROM    SYSTABLES
WHERE   COLCOUNT>5
```

Esta consulta SQL busca en el catálogo los nombres de las tablas que contienen más de cinco columnas.

El nombre del creador de las tablas del catálogo es SYSIBM; por tanto, hacemos referencia al nombre completo de una tabla, como SYSTABLES, escribiendo SYSIBM.SYSTABLES. El sistema crea automáticamente entradas de catálogo para las tablas del catálogo. Los usuarios autorizados tienen acceso al catálogo para consultarlo. Así, quienes tienen privilegio de SELECT para el catálogo del sistema, si no están familiarizados con la estructura de la base de datos, pueden consultar el catálogo para conocerla mejor.

Por ejemplo, la consulta

```
SELECT  TBNAME
FROM    SYSIBM.SYSCOLUMNS
WHERE   NAME='NÚMEROD'
```

lista los nombres de las tablas DEPARTAMENTO y LUGARES_DEPTOS porque contienen la columna NÚMEROD (véase la Fig. 6.5). La disponibilidad para el usuario de la misma interfaz SQL para tener acceso a los metadatos, y no sólo a los datos, es un recurso importante de DB2.

Actualización de la información del catálogo. Si bien para los usuarios la consulta del catálogo es informativa, su actualización puede ser realmente devastadora. Por ejemplo, una solicitud corriente de actualización como

```
DELETE
FROM    SYSIBM.SYSTABLES
WHERE   CREATOR=NAVATHE
```

eliminará del catálogo las entradas de todas las tablas creadas por NAVATHE. En consecuencia, dejan de existir las definiciones de dichas tablas, aunque todavía existan las tablas mismas. En esencia, las tablas se han vuelto inaccesibles. A fin de evitar tales situaciones, *no* está permitido realizar las operaciones UPDATE, DELETE ni INSERT en las tablas del catálogo; estas funciones se llevan a cabo con ALTER TABLE, DROP TABLE y CREATE TABLE, que son las instrucciones de definición de datos en SQL. El enunciado COMMENT (comentario) de SQL es útil porque permite almacenar información textual sobre una tabla o columna del catálogo.

Por ejemplo, con referencia a la definición de base de datos de la figura 7.1, el enunciado "COMMENT ON COLUMN DEPENDIENTE.NSSE IS 'Si ambos padres de un dependiente son empleados, el dependiente se representa dos veces.'" se almacena en las columnas REMARKS apropiadas de la entrada en la tabla SYSCOLUMNS.

9.4 Manipulación de datos en DB2

SQL es el lenguaje de manipulación de datos primordial de DB2. En el capítulo 7 examinamos una versión de SQL que corresponde muy de cerca a su implementación en DB2. Como referencia rápida, ofrecemos la siguiente lista con algunos tipos de obtenciones y actualizaciones que permite realizar el SQL de DB2 (los números se refieren a los ejemplos de consultas del capítulo 7); las versiones más nuevas de DB2 tal vez manejen algunas de las características que por ahora no están disponibles:

1. Obtenciones simples de tablas (C0).
2. Presentar las filas completas de la tabla que satisfacen una condición previamente especificada (C1C).
3. Obtenciones con eliminación de filas repetidas (C11A).
4. Obtenciones con valores calculados a partir de las columnas (C15).
5. Ordenación del resultado de una consulta (C28).
6. Obtenciones con condiciones que implican conjuntos e intervalos. Éstas se realizan con diversos tipos de constructores:
   a. Con IN (C13).
   b. Con BETWEEN (entre; éste no lo vimos en el capítulo 7): DB2 permite construcciones con BETWEEN o NOT BETWEEN; p. ej., "WHERE SALARIO BETWEEN 50 000 AND 100 000".
   c. Con LIKE (C25, C26).
   d. Con NULL en comparaciones (C14).

7. Obtenciones a partir de múltiples tablas mediante reuniones (C1, C2, C8, etcétera).
8. Consultas anidadas: La anidación puede lograrse pasando los resultados de la subconsulta interior a la consulta exterior, mediante IN (véase C12, C4A). En general, las consultas pueden anidarse hasta cualquier número de niveles. Como se ilustra en C4A, las subconsultas interior y exterior pueden referirse a la misma tabla.
9. Con EXISTS: La cuantificación existencial (Cap. 8) se logra conectando dos subconsultas con EXISTS (C12B, C7). Como no es posible aplicar directamente la cuantificación universal, ello se hace con NOT EXISTS (C6, C3A).
10. Empleo de funciones integradas: las funciones del SQL de DB2 incluyen COUNT, SUM, AVG, MAX y MIN. EXISTS también se considera una función integrada, aunque, en vez de devolver un valor numérico o de cadena, devuelve un valor lógico.
11. Agrupación (C20, C21) y condiciones sobre grupos (C22).
12. UNIÓN: Es posible obtener la unión de los resultados de subconsultas, siempre que sean compatibles con la unión (véase la definición en el capítulo 6). En DB2 La posibilidad de incluir cadenas en la instrucción SELECT resulta muy útil junto con UNIÓN. Por ejemplo, si queremos obtener una lista de las personas que trabajaron más de 40 horas o que trabajaron en el proyecto P5, podríamos introducir

```
SELECT   NSSE, 'trabajó más de 40 horas.'
FROM     TRABAJA_EN
WHERE    HORAS>40
UNION    SELECT   NSSE, 'trabajó en el proyecto P5'
         FROM     TRABAJA_EN
         WHERE    NÚMP = P5
```

Los resultados podrían verse así:

```
1000 trabajó más de 40 horas.
1002 trabajó más de 40 horas.
1003 trabajó más de 40 horas.
1002 trabajó en el proyecto P5.
1007 trabajó en el proyecto P5.
```

Observe que, si no hubiéramos incluido restricciones de cadena en la cláusula SELECT, la entidad repetida (1002) habría sido eliminada.

13. CONTAINS, INTERSECTION y MINUS. El SQL de DB2 *no* cuenta con estos operadores. Se deben sustituir por construcciones con EXISTS y NOT EXISTS. Además, DB2 *no* maneja la unión externa (OUTER UNION).
14. Inserción (A1, A1A, A3B).
15. Eliminación: Se logra con DELETE (A4A-A4D). Si queremos eliminar por completo la definición de la tabla debemos usar DROP TABLE.
16. Modificación: Se logra con UPDATE (A5, A6).

Las instrucciones de actualización del SQL de DB2 tienen el defecto de que cuando en un INSERT, DELETE o UPDATE interviene una subconsulta, la subconsulta anidada no puede hacer referencia a la tabla destino de la operación. Por ejemplo, si queremos actualizar el valor de HORAS en todas las filas, asignándole el promedio de las horas, sería deseable un tipo de construcción como el que sigue, pero que *no* está permitido en DB2:

```
UPDATE   TRABAJA_EN
SET      HORAS=X
WHERE    X = (  SELECT   AVG(HORAS)
                FROM     TRABAJA_EN )
```

(Una construcción que hiciera innecesaria X eliminando la cláusula WHERE y anidara un SELECT dentro del UPDATE funcionaría aún mejor.) En la práctica, el resultado se logra obteniendo primero el promedio y empleando después ese promedio para hacer la actualización.

DB2 Version 2 Release 3 (y Version 3) se ajustan a las especificaciones ISO/ANSI SQL89; por ello, es posible que algunas de las siguientes características de SQL2, que vimos en el capítulo 7, no estén disponibles en DB2:

1. La orden CREATE DOMAIN (como mecanismo para definir nuevos tipos que sirvan para referirse a conjuntos de valores).
2. Empleo de tablas reunidas dentro de la cláusula FROM (Sec. 7.2.8).
3. La orden CREATE ASSERTION (Sec. 7.5) y las instrucciones TRIGGER con procedimientos de acción.

9.4.1 *Procesamiento de vistas*

Como vimos en la sección 7.4, se puede definir una vista en DB2 con la instrucción CREATE VIEW AS seguida de una consulta SQL. Debemos tener presentes los siguientes puntos al definir vistas en DB2:

- En una definición de vista pueden intervenir una o más tablas; puede incluir reuniones y funciones integradas.
- Si no se especifican nombres de columnas en la definición de vista, se pueden asignar automáticamente, excepto cuando se utilicen funciones integradas, expresiones aritméticas o restricciones.
- La consulta SQL con que se define la vista no puede usar UNION ni ORDER BY.
- Es posible definir vistas sobre vistas ya existentes.
- Cuando se define una vista con la cláusula WITH CHECK OPTION, cualquier INSERT o UPDATE en términos de esa vista se someterá a comprobación para confirmar que, en efecto, se satisfaga el predicado de definición de la vista. Por ejemplo, en la instrucción siguiente el predicado de definición de la vista es SALDO > 500:

```
CREATE VIEW      SUJETO_DE_CRÉDITO
AS  SELECT       NÚMCLI, NOMBRE, NÚMTEL
    FROM         CLIENTE
    WHERE        SALDO>500
WITH CHECK OPTION
```

Obtenciones de datos de vistas. Al especificar consultas para obtención de datos las vistas se tratan igual que las tablas base. Puede haber problemas cuando un atributo de la vista sea resultado de una función integrada aplicada a una tabla base subyacente. Por ejemplo, consideremos la siguiente definición de vista:

```
CREATE VIEW   RESUMEN_DEPTO(NDEP, SALARIOTOTAL)
AS            SELECT ND, SUM(SALARIO)
              FROM EMPLEADO
              GROUP BY ND
```

Veamos ahora dos consultas. En primer lugar, la consulta

```
SELECT   NDEP
FROM     RESUMEN_DEPTO
WHERE    SALARIOTOTAL > 100 000
```

no es válida, porque después de la conversión la cláusula WHERE quedará como WHERE SUM(SALARIO) > 100 000, y *no* pueden incluirse funciones integradas en una cláusula WHERE. La conversión correcta (¿puede escribirla el lector?) contiene una cláusula HAVING, pero en DB2 no es posible una consulta convertida como esa. En segundo lugar, la consulta

```
SELECT   SUM(SALARIOTOTAL)
FROM     RESUMEN_DEPTO
```

tampoco es válida, porque SUM(SALARIOTOTAL) equivale a SUM(SUM(SALARIO)), y DB2 no permite anidar las funciones integradas.

Actualización de vistas. Ya examinamos en la sección 7.4.3 los problemas de la actualización de vistas. DB2 no cuenta con recursos para investigar lo que un usuario desea hacer cuando especifica una actualización de vista. Por añadidura, no hay mecanismos para analizar una actualización y determinar si equivale a un conjunto único de actualizaciones de las relaciones base. Por tanto, DB2 adopta un enfoque más bien restringido al permitir únicamente la actualización de vistas de una sola relación. Además, *incluso en el caso de vistas de una sola relación*, valdrán las siguientes restricciones:

- No se puede actualizar una vista si (a) en su definición interviene una función integrada, (b) su definición tiene DISTINCT en la cláusula SELECT, (c) su definición incluye una subconsulta y la cláusula FROM de dicha subconsulta hace referencia a la tabla base sobre la cual está definida la vista, o (d) hay una cláusula GROUP BY en la definición de la vista.
- Si un campo de la vista se deriva de una expresión aritmética o de una constante, *no* se permiten inserciones ni actualizaciones; sin embargo, sí se permite DELETE (ya que es posible eliminar una fila correspondiente de la tabla base).

9.4.2 *Empleo de SQL incorporado*

En la sección 7.7 ya tratamos con detalle el SQL incorporado. Aquí mencionaremos unos cuantos puntos más referentes al empleo de SQL incorporado en DB2. Como vimos, es posible

incorporar SQL en programas escritos en COBOL, PL/1, FORTRAN, C, PROLOG y lenguaje ensamblador. Las pautas de incorporación son las siguientes:

- Cualesquiera tablas base o vistas que utilice el programa deberán declararse por medio de un enunciado DECLARE. Esto facilita la comprensión del programa y ayuda al precompilador a verificar la sintaxis.
- Las instrucciones de SQL incorporado deben ir precedidas por EXEC SQL y pueden ir en cualquier lugar en que esté permitida una instrucción *ejecutable* del lenguaje anfitrión.
- El SQL incorporado puede incluir recursos de definición de datos, como CREATE TABLE y DECLARE CURSOR, que son puramente declarativos.
- Las instrucciones de SQL pueden hacer referencia a variables del lenguaje anfitrión precedidas por un signo de dos puntos.
- Las variables anfitrión que reciban valores de SQL deberán tener tipos de datos compatibles con las definiciones de campos en SQL. La compatibilidad no se define con mucha rigurosidad; por ejemplo, las cadenas de caracteres de diversas longitudes o los datos numéricos de naturaleza binaria o decimal se consideran compatibles. DB2 se encarga de las conversiones apropiadas.
- El área de comunicación de SQL (SQL Communication Area: SQLCA) actúa como área de retroalimentación común entre el programa de aplicación y DB2. Un indicador de estado SQLCODE contiene un valor numérico que muestra el resultado de una consulta (por ejemplo, cero indica que se completó con éxito, y +100 indica que se ejecutó la consulta pero que el resultado es nulo).
- No se necesita un cursor para una consulta de obtención de datos SQL que devuelva una sola tupla, ni para las instrucciones UPDATE, DELETE o INSERT (excepto cuando se requiera el CURRENT OF de un registro: véase la Sec. 7.7).
- Puede usarse un programa de utilería especial llamado DCLGEN (generador de declaraciones) para construir automáticamente enunciados DECLARE TABLE en PL/1 a partir de las definiciones CREATE TABLE en SQL. También se generan automáticamente estructuras PL/1 o COBOL que corresponden a las definiciones de las tablas.
- La instrucción WHENEVER (siempre que), colocada fuera de línea, permite revisar SQLCODE para detectar una situación específica. En los ejemplos

```
WHENEVER   NOTFOUND PERFORM X;
WHENEVER   SQLERROR GO TO Y;
```

NOTFOUND y SQLERROR son palabras reservadas del sistema que corresponden a SQLCODE = 100 y SQLCODE = otro valor que no sea 0 ni 100, respectivamente.

9.4.3 *Integridad referencial*

Definimos el concepto de integridad referencial en la sección 6.2.4. Se basa en la noción de claves externas de una relación, que dependen de las claves primarias de otras relaciones. En términos sencillos, la restricción de integridad especifica que la clave externa puede ser

nula o bien tener un valor que se refiera a un valor válido *ya existente* como valor de clave primaria en alguna otra tabla. Si durante una actualización se viola esta restricción, el SGBDR deberá rechazar la actualización o bien emprender una *acción correctiva*. Estas acciones se examinaron en la sección 6.3.

La orden CREATE TABLE que presentamos en la sección 7.1.2, junto con las especificaciones PRIMARY KEY y FOREIGN KEY, se aplican en DB2. DB2 permite designar claves primarias y externas en las órdenes CREATE TABLE y ALTER TABLE. También estudiamos las opciones SET NULL y CASCADE, con las que se puede indicar la acción a realizar cuando se viola la integridad referencial (véase el esquema de la figura 7.1). Éstas son aplicables en DB2. También hay una opción RESTRICT en DB2. Por ejemplo, si queremos evitar la eliminación de un departamento al que pertenezca actualmente algún empleado, podemos modificar la instrucción CREATE TABLE EMPLEADO de la figura 7.1(a) de modo que contenga la siguiente especificación:

FOREIGN KEY (ND) **REFERENCES** DEPARTAMENTO (NÚMEROD) **ON DELETE RESTRICT;**

DB2 permite declarar las claves primarias NOT NULL o NOT NULL WITH DEFAULT (no nulas con valor por omisión). La segunda especificación permite usar un espacio en blanco o cero como valor de clave primaria. Si se declara un índice único sobre la columna de la clave primaria y se especifica NOT NULL WITH DEFAULT, sólo se permitirá una clave con el valor cero o blanco.

Ciertas reglas se aplican en DB2 a la inserción, eliminación o modificación de valores de clave externa, así como a la eliminación o modificación de valores de clave primaria. Cabe señalar que la eliminación de una clave externa (asignándole el valor nulo) o la inserción de un valor de clave primaria *no* provocan la violación de una restricción de integridad. Las reglas pueden resumirse como sigue:

1. Un valor de clave primaria puede modificarse si no corresponde a ningún valor de clave externa.
2. Si un usuario intenta eliminar un valor de clave primaria, y existe un valor de clave externa correspondiente, entonces:
   a. La eliminación está prohibida si la restricción de clave externa es RESTRICT.
   b. Las tuplas correspondientes (en otras tablas) con valores de clave externa idénticos se eliminan si la especificación de restricción es CASCADE.
   c. Se asigna NULL a los valores de clave externa correspondientes si la especificación de restricción es SET NULL.
3. La inserción o modificación de un valor de clave externa sólo están permitidas si existe un valor correspondiente de clave primaria.

El que las restricciones de integridad referencial se ajusten a las reglas anteriores implica que el sistema realizará un procesamiento interno con las operaciones de E/S, los bloqueos de control de concurrencia, etc., que sean necesarios. En general el sistema usa índices, si cuentan con ellos, para la verificación, y realiza ésta en el nivel del manejador de datos sin pasar los datos al "nivel superior" del sistema llamado Relational Data System (sistema de datos relacional). Así, el proceso tiende a ser más eficiente que si lo efectuara una aplicación. Para las operaciones por lotes en que se elimina o inserta un gran número de filas,

puede resultar más eficiente que la aplicación imponga las restricciones de integridad referencial emprendiendo una acción correctiva apropiada a gran escala, en vez de dejar que el sistema compruebe si la restricción se aplica a la eliminación o la inserción en cada fila.

Las utilerías como LOAD (véase la Sec. 9.5.3) proveen un mecanismo para desactivar la comprobación de restricciones. Para hacer esto durante las actualizaciones por lotes, podemos usar el recurso ALTER TABLE para desechar temporalmente las especificaciones de clave externa; y después de la actualización podemos asegurarnos de que los datos sean consistentes mediante la utilería CHECK DATA.

El catálogo mantiene información de integridad referencial en diversos sitios, como los siguientes:

- SYSFOREIGNKEYS contiene el nombre de restricción de clave externa llamada RELNAME y las columnas que contienen la clave.
- SYSCOLUMNS.KEYSEQ, FOREIGNKEY indica si la columna es parte de una clave primaria o de una clave externa.
- SYSTABLESPACE.STATUS indica si el espacio de tablas está en situación de verificación pendiente respecto a la comprobación de la validez de las restricciones de integridad.

Los distintos SGBDR proveen especificación e imposición de la integridad referencial en diferentes niveles de detalle. En algunos sistemas se deja que la aplicación proporcione estas funciones.

9.5 Almacenamiento de datos en DB2★

Una base de datos en DB2 es una colección de objetos que guardan relaciones lógicas entre sí. Estos objetos son las diversas tablas e índices almacenados físicamente. DB2 utiliza una terminología especial para describir las áreas de almacenamiento delimitadas. **Espacio de tablas** (*tablespace*) se refiere a la parte del almacenamiento secundario donde se guardan las tablas, y **espacio de índices** (*indexspace*) se refiere a la parte donde se almacenan los índices. La figura 9.4 es un esquema de la estructura de almacenamiento de DB2. Se muestra la colección total de datos de un sistema formado por las bases de datos de usuario BDX y BDY y la base de datos del catálogo del sistema.

La **página** es la unidad básica de transferencia de datos entre el almacenamiento secundario y el primario. Un **espacio** es una colección de páginas dinámicamente extensible, y cada espacio pertenece a un **grupo de almacenamiento**, que es una colección de áreas de almacenamiento de acceso directo del mismo tipo de dispositivo. No existe correspondencia 1:1 entre una base de datos y un grupo de almacenamiento. En la figura 9.4, el mismo grupo de almacenamiento contiene un espacio de índices y un espacio de tablas de la base de datos BDX y un espacio de tabla de la base de datos BDY. Una **base de datos** es una unidad de **inicio/paro** (*start/stop*) que el operador de la consola habilita o inhabilita por medio de una orden START o STOP. Las tablas se pueden pasar de una base de datos a otra sin afectar a los usuarios ni sus programas.

Los grupos de almacenamiento se controlan empleando archivos VSAM (conjuntos de datos en el orden en que se introdujeron). Dentro de una página, DB2 controla la reorganización sin usar VSAM para nada. DB2 permite al DBA y al administrador del sistema especificar los detalles de la estructura de almacenamiento, usando diversas instrucciones. Para cada objeto descrito (como una tabla, un espacio de tablas, un índice, un espacio de índices, una

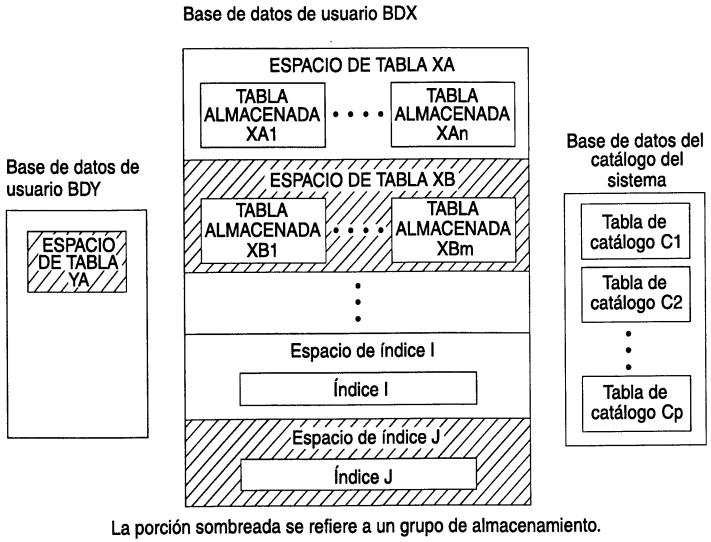


Figura 9.4 Esquema de la estructura de almacenamiento de DB2.

base de datos o un grupo de almacenamiento), las tres instrucciones que se utilizan uniformemente son CREATE (crear), ALTER (alterar) y DROP (desechar). Los usuarios *no* necesitan conocer la organización interna del almacenamiento para poder emplear el sistema. Aunque se requiere un espacio de tablas para almacenar una tabla, el sistema asigna un espacio *por omisión* cuando el creador de la tabla no lo especifica. Una base de datos es una entidad lógica cuyos componentes físicos se pueden cambiar de lugar y manipular libremente sin afectar la integridad de la base de datos. La orden CREATE TABLESPACE identifica la base de datos a la que pertenece el espacio de tablas.

9.5.1 *Espacios de tablas y tablas almacenadas*

El espacio de tablas para una tabla dada se especifica en la instrucción CREATE TABLE de esa tabla. El tamaño de página de un espacio de tabla es 4096 o bien 32 768 bytes. Un espacio de tablas puede crecer si se añade más almacenamiento de un grupo de almacenamiento, hasta un límite superior de 64 gigabytes. El espacio de tablas es la unidad de almacenamiento sujeta a reorganización o recuperación por efecto de una orden de la consola. Como sería muy poco eficiente manejar un espacio de tablas grande de esta manera, DB2 permite dividir en particiones los espacios de tablas.

Un espacio de tablas sin particiones se denomina **espacio de tablas simple**. En la mayoría de los casos, un espacio de tablas simple contiene una tabla. Podemos almacenar varias tablas —digamos, EMPLEADO y DEPARTAMENTO— en el mismo espacio de tablas para mejorar el rendimiento si es alta la probabilidad de obtener acceso a ambas a la vez. El índice de una tabla, si lo hay, se guarda en un espacio de índices. Una tabla con índice de agrupamiento (véase el Cap. 5) se carga inicialmente en el espacio de tablas en orden según

la clave, empleando una utilería de carga. Se incluyen huecos intermitentes para contener los registros que se inserten posteriormente. Si no hay un índice de agrupamiento, los registros pueden cargarse en cualquier orden y las inserciones se realizan al final del archivo.

Un **espacio de tablas con particiones** contiene una tabla dividida en (agrupada por) intervalos de valores de uno o más campos de partición. Es *obligatorio* tener un índice de agrupamiento sobre esos campos, y dicho índice no puede modificarse. Así, la tabla EMPLEADO puede dividirse mediante un índice de agrupamiento sobre ND. La ventaja de los espacios de tablas con particiones es que cada partición se trata como objeto de almacenamiento individual para fines de recuperación y reorganización, por lo que puede estar asociada a un diferente grupo de almacenamiento.

La división de un espacio de tablas ofrece varias ventajas en el caso de tablas grandes:

- *Mayor disponibilidad de los datos*: Un usuario puede realizar el mantenimiento normal en una partición de la tabla mientras el resto sigue estando disponible para su procesamiento con utilerías o SQL.
- *Mejor desempeño de las utilerías*: Una tarea de utilería puede trabajar con todas las particiones simultáneamente, en vez de trabajar con ellas una por una. Esto puede reducir significativamente el tiempo requerido para completar una tarea de utilería.
- *Menor tiempo de respuesta de las consultas*: Cuando DB2 examina los datos para responder a una consulta, en ocasiones puede examinar varias particiones al mismo tiempo, en vez de examinar todo el espacio de tablas de principio a fin. Esta mejoría es significativa sobre todo en el caso de consultas complejas o que obligan a DB2 a examinar grandes cantidades de datos.

Los **espacios de tablas segmentados** están diseñados para contener más de una tabla. El espacio disponible se divide en grupos de páginas llamados *segmentos*, todos del mismo tamaño. Cada segmento contiene filas de una sola tabla. Si queremos buscar en todas las filas de una tabla, no es necesario examinar todo el espacio de tablas, sino sólo los segmentos que contienen esa tabla. Si se desecha una tabla, sus segmentos quedan de inmediato disponibles para su reutilización. Todos los segmentos deben residir en el mismo conjunto de datos definido por el usuario o en el mismo grupo de almacenamiento.

Cada fila de una tabla constituye un **registro almacenado**: una cadena de bytes que comprende un prefijo con información de control del sistema y hasta *n* campos almacenados, donde *n* es el número de columnas de la tabla base. Los campos nulos al final de un registro de longitud variable *no* se almacenan. Internamente, cada registro tiene un **identificador de registro** (RID) único dentro de una base de datos; consiste en el número de página y el desplazamiento en bytes, respecto al principio de la página, de una caja que, a su vez, contiene la posición de inicio del registro dentro de la página.

Cada **campo almacenado** incluye tres elementos:

- Un campo de prefijo que contiene la longitud de los datos, si es variable.
- Un prefijo indicador de nulo que indica si el campo contiene un valor nulo.
- Un valor de datos codificado.

DB2 ha adoptado la estrategia de almacenar todos los tipos de datos de tal manera que se les considere como cadenas de bytes y que la instrucción de "comparación lógica"

siempre produzca un resultado correcto (incluso con el tipo de datos entero, INTEGER). La interpretación de las cadenas de bytes no es tarea del manejador de datos almacenados. Los campos variables ocupan sólo el espacio real requerido. La compresión o el cifrado de los datos se dejan abiertas a un procedimiento provisto por el usuario, que puede interponerse cada vez que se lea o grabe un registro almacenado. Los registros *dentro de una página* pueden reorganizarse sin alterar sus RID.

9.5.2 *Espacios de índices e índices*

Un espacio de índices corresponde al almacenamiento ocupado por un índice. A diferencia del espacio de tablas, se crea automáticamente. Las páginas de los índices tienen 4096 bytes de largo, pero pueden bloquearse por cuartos de página. Un espacio de índices que contiene un índice de agrupamiento para un espacio de tablas dividido se considera que también está dividido en particiones.

Los índices son árboles B^+ (véase el Cap. 5) en los que cada nodo es una página. Las páginas hoja están encadenadas a fin de proveer acceso secuencial a las filas (en el espacio de tablas). Una tabla puede tener un índice de agrupamiento y cualquier cantidad de índices no agrupados. Las páginas hoja de un índice no agrupado tienen acceso a las filas del espacio de tablas en un orden diferente al de su orden físico; por tanto, para lograr el procesamiento secuencial eficiente de una tabla es indispensable contar con un índice de agrupamiento.

9.5.3 *Utilerías de DB2*

DB2 ofrece diversas utilerías con las que los creadores de aplicaciones pueden manipular los datos almacenados sin necesidad de escribir programas individuales. En seguida proporcionamos una lista de algunas de las utilerías para manejar los "objetos de datos" de DB2 (según la terminología de DB2):

- **LOAD**: La utilería LOAD carga datos en tablas DB2 a partir de conjuntos de datos cuyo método de acceso es el secuencial básico (BSAM: *basic sequential access method*), tablas SQL/DS no cargadas y tablas DB2 no cargadas. La partición de un espacio de tablas puede reemplazarse mediante la instrucción LOAD REPLACE. La utilería de carga garantiza que los datos cargados en la tabla y en el índice sean consistentes y utilizables.
- **REORG**: La utilería REORG reorganiza los espacios de tablas y de índices. Puede restablecer el espacio libre y la secuencia física descrita por un índice de agrupamiento. Recupera el espacio perdido por fragmentación y por desechar tablas.
- **CHECK**: La utilería CHECK INDEX prueba si los índices son congruentes con los datos que indizan y emite mensajes de aviso cuando detecta una inconsistencia. La utilería CHECK DATA busca violaciones de la integridad referencial e indica el lugar donde ocurren; puede servir para eliminar violaciones de la integridad referencial.
- **STOSPACE**: La utilería STOSPACE provee información acerca del empleo actual del espacio por parte de los espacios de tablas y de índices en un grupo de almacenamiento DB2 dado. Esta utilería puede actualizar el catálogo de DB2 con dicha información.

- **RUNSTATS**: La utilería RUNSTATS proporciona información estadística referente a tablas, espacios de tablas y espacios de índices, y puede actualizar el catálogo de DB2 con esta información. DB2 utiliza estos datos estadísticos para optimizar el rendimiento de las instrucciones SQL (analizaremos este punto al tratar la optimización de consultas en el capítulo 16). Los administradores de bases de datos pueden usarlos para evaluar la situación de un espacio de tablas o de índices determinado.
- **REPAIR**: La utilería REPAIR repara los datos. Éstos pueden ser los propios datos del usuario o datos a los que el usuario normalmente no tendría acceso de manera explícita, como es el caso de las entradas de índice.
- **Utilerías para diagnóstico**: DB2 cuenta también con una variedad de utilerías que ayudan a los equipos creadores de aplicaciones a diagnosticar los problemas. Entre otras cosas, estas utilerías pueden crear vaciados de memoria con base en sucesos de DB2, comprobar la integridad de los espacios de tablas del catálogo y exhibir el contenido de las bitácoras de recuperación.

9.6 Características internas de DB2★

En esta sección resumiremos las características de DB2 relacionadas con la seguridad, la autorización y el procesamiento de transacciones.

9.6.1 *Seguridad y autorización*

En general, la seguridad de los datos se cuida en dos niveles en DB2:

1. El mecanismo de vistas puede servir para ocultar datos confidenciales a usuarios no autorizados.
2. El subsistema de autorización, que proporciona privilegios específicos a ciertos usuarios, les permite otorgar dichos privilegios a otros usuarios de manera selectiva y dinámica, y revocarlos a voluntad.

Las órdenes GRANT (otorgar) y REVOKE (revocar) y la característica GRANT OPTION se analizarán en la sección 20.2. Aquí sólo plantearemos algunas consideraciones específicas en torno a estas órdenes que son aplicables en DB2. El DBA o un administrador apropiado es el encargado de tomar las decisiones referentes al otorgamiento de privilegios específicos a los usuarios y a la revocación de dichos privilegios. Esta decisión de política puede comunicarse a DB2 mediante instrucciones CREATE VIEW o GRANT y REVOKE. Esta información reside en el catálogo del sistema. Compete al sistema imponer estas decisiones en el momento de la ejecución cuando se intenten operaciones de obtención de datos o de actualización; esta función la realiza el componente enlazador (véase la Fig. 9.2).

Identificación del usuario. Para que DB2 conozca a los usuarios legítimos interviene un identificador de autorización llamado AUTHID, asignado por los administradores del sistema. El usuario está obligado a usar dicho identificador al entrar al sistema. Los usuarios de DB2 entran primero a CICS, a IMS o a TSO, y el subsistema en cuestión pasa el identificador a DB2; así pues, la obligación de verificar el identificador recae en uno de esos subsistemas.

La palabra reservada USER se refiere a una variable del sistema cuyo valor es un identificador de autorización. Si un usuario en particular está usando una vista (para obtener datos o para actualizarlos), la variable USER contiene el identificador del usuario que está *utilizando* la vista, *no* el del que la creó. Así,

```
CREATE  VIEW   TABLAS_PROPIAS
AS             SELECT  *
               FROM    SYSIBM.SYSTABLES
               WHERE   CREATOR=USER
```

es una definición de vista que selecciona las tablas creadas por el usuario que ingresó al sistema. SYSIBM.SYSTABLES y CREATOR son palabras reservadas en DB2. Si un usuario cuyo identificador es sbn134 ingresó al sistema y ejecuta la consulta

```
SELECT  *
FROM    TABLAS_PROPIAS
```

la variable USER de la vista se enlaza a sbn134, y el resultado de la consulta es la obtención de las entradas de SYSTABLES que fueron creadas por sbn134.

Las vistas como mecanismo de seguridad. Es posible usar vistas para fines de seguridad si se oculta a los usuarios no autorizados los datos que no deben ver. Si escoge las condiciones apropiadas en la cláusula WHERE, e incluye en la cláusula SELECT sólo las columnas que el usuario tiene permiso de ver, el diseñador del sistema puede ocultar ciertos datos a ese usuario. Las vistas definidas sobre información del catálogo del sistema, como la que acabamos de ilustrar, permiten que un usuario sólo vea ciertas partes del catálogo. Si aplica funciones agregadas como SUM y AVG, el diseñador puede permitir que un usuario vea un resumen estadístico de la tabla base, pero no los valores individuales.

En DB2, cuando se inserta o actualiza un registro a través de una vista, *no* es obligatorio que la fila nueva o modificada deba obedecer las condiciones o predicados de definición de la vista. En ocasiones esto puede originar una situación en la que el registro nuevo o actualizado desaparezca de inmediato de la vista del usuario, aunque aparecerá en la tabla base subyacente. Para impedir tales inserciones o eliminaciones, debe usarse CHECK OPTION en la definición de la vista.

Mecanismos de concesión y revocación. Las instrucciones GRANT y REVOKE de SQL determinan las operaciones específicas que un usuario puede o no realizar. (Aquí es válida la explicación general de la sección 20.2.) Los privilegios concedidos (por GRANT) a los usuarios se pueden clasificar en las siguientes categorías amplias:

- Privilegios de tablas y vistas: Se aplican a las tablas base y a las vistas.
- Privilegios de base de datos: Se aplican a las operaciones con una base de datos (como la creación de una tabla).
- Privilegios de plan de aplicación: Se refieren a la ejecución de planes de aplicación.
- Privilegios de almacenamiento: Tienen que ver con el empleo de ciertos objetos de almacenamiento, como son espacios de tablas, grupos de almacenamiento y reservas de almacenamiento intermedio.

- Privilegios de sistemas: Se aplican a operaciones del sistema (como la creación de una nueva base de datos).

También existen ciertos privilegios "envueltos", término que se refiere a una combinación personalizada de privilegios:

- El privilegio SYSADM (administrador del sistema) es el de más alto orden e incluye todos los posibles privilegios del sistema.
- El privilegio DBADM (administrador de la base de datos) sobre una base de datos específica permite al poseedor ejecutar cualquier operación con esa base de datos.
- El privilegio DBACTRL (control de base de datos) sobre una base de datos específica es similar a DBADM, excepto que sólo se permiten operaciones de control, no de manipulación de datos (por ejemplo, en SQL).
- El privilegio DBMAINT (mantenimiento de base de datos) sobre una base de datos específica permite al poseedor ejecutar operaciones de mantenimiento sólo de lectura (como la preparación de copias de seguridad) con la base de datos. Es un subconjunto del privilegio DBACTRL.
- El privilegio SYSOPR (operador del sistema) permite al poseedor efectuar exclusivamente funciones de operador de consola, sin acceso a la base de datos.

Un identificador de autorización tiene el privilegio SYSADM, y representa la función del administrador del sistema. Otros identificadores pueden poseer el privilegio SYSADM, pero éste puede revocarse. PUBLIC es una palabra reservada del sistema e incluye todos los identificadores de autorización.

He aquí unas notas finales respecto a cómo se implementan estas características en DB2:

- Un beneficio importante en cuanto al rendimiento se debe al hecho de que es posible aplicar muchas comprobaciones de autorización en el momento del enlace (compilación), y no hay que esperar hasta el momento de la ejecución.
- DB2 trabaja con diversos sistemas acompañantes; además de ellos, proporciona seguridad en el sistema. Los mecanismos de control individuales de MVS, VSAM, IMS y CICS ofrecen protección adicional.
- Toda la gama de mecanismos de autorización y seguridad es opcional; por tanto, es posible inhabilitarlos, permitiéndose así que cualquier usuario tenga privilegios de acceso totales.

9.6.2 *Procesamiento de transacciones*

Estudiaremos el concepto de transacción y los problemas de recuperación y control de concurrencia relacionados con el procesamiento de transacciones en los capítulos 17 a 19. Aquí sólo describiremos las características específicas de DB2 y SQL. Tal vez los lectores interesados deseen leer los conceptos descritos en los capítulos mencionados antes de seguir con el resto de esta sección.

En primer lugar, modifiquemos el ejemplo E2 de SQL incorporado de la sección 7.7 para mostrar cómo se escribe una transacción real en DB2. Usaremos PL/1 aquí, pues DB2 no

acepta PASCAL. La transacción consiste en otorgar un aumento *A* a un empleado cuyo número de seguro social es *S*:

```
TRANS1: PROC OPTIONS (MAIN);
EXEC SQL WHENEVER SQLERROR GO TO PROC_ERROR;
DCL S FIXED DECIMAL (9.0);
DCL A FIXED DECIMAL (7.2);
GET LIST (S, A);
EXEC SQL UPDATE EMPLEADO
SET SALARIO = SALARIO + :A
WHERE NSS = :S;
EXEC SQL WHENEVER NOTFOUND GO TO IMPR_MENSAJE;
COMMIT;
GO TO SALIR;
IMPR_MENSAJE: PUT LIST ('EL EMPLEADO', S, 'NO ESTÁ EN LA BASE DE DA-
TOS'); GO TO SALIR;
PROC_ERROR: ROLLBACK;
SALIR: RETURN;
END TRANS1;
```

En este ejemplo, la transacción puede fallar si ningún empleado tiene el número de seguro social *S* (NOTFOUND) o si SQLCODE devuelve un valor negativo (SQLERROR).

También podemos insertar verificaciones en el programa, como sería el caso para asegurarnos de que el salario no sea nulo, pues sin ello la transacción podría fracasar. La operación COMMIT (confirmación) señala que la transacción terminó con éxito; ordena al gestor de transacciones confirmar la modificación de la base de datos: esto es, hacer que sea permanente, dejando la base de datos en un estado consistente. La operación ROLLBACK (reversión), por su parte, señala que la transacción no se completó con éxito; le ordena al gestor de transacciones *no* hacer cambios permanentes en la base de datos, y dejarla en el estado en que se encontraba antes de iniciarse la ejecución de esta transacción.

Una transacción ordinaria puede implicar varias obtenciones de datos y actualizaciones; sin embargo, *sólo hay una* operación COMMIT en el programa, de modo que o se aplican *todos* los cambios, o no se aplica *ninguno*. La operación ROLLBACK utiliza las entradas de la bitácora y restaura, según sea apropiado, los elementos actualizados a sus valores previos.

Toda operación DB2 se ejecuta en el contexto de alguna transacción. Esto *incluye* las que se introducen interactivamente a través de DB2I. Una aplicación consiste en una serie de transacciones. Las transacciones *no pueden* anidarse unas en otras.

Confirmación y reversión en DB2. El SGBD DB2 está *subordinado* al gestor de transacciones (IMS, CICS o TSO) bajo el cual se ejecuta. Actúa como uno de los gestores de recursos al proveer un servicio al gestor de transacciones. Por tanto, debemos tener presentes los siguientes puntos:

- COMMIT y ROLLBACK no son operaciones de la base de datos; son instrucciones para el gestor de transacciones, que no forma parte del SGBD.
- Si una transacción actualiza una base de datos IMS y una base de datos DB2, todas las actualizaciones (tanto a IMS como a DB2) deben confirmarse o bien todas deben revertirse.

- Un "punto de sincronización" define un punto en el que la base de datos está en un estado consistente. El inicio de una transacción, COMMIT y ROLLBACK establecen un punto de sincronización; ninguna otra cosa lo hace.
- La operación COMMIT señala que la transacción concluyó con éxito, establece un punto de sincronización, confirma todas las actualizaciones de la base de datos realizadas desde el punto de sincronización previo, cierra todos los cursores abiertos y libera todos los bloqueos (con algunas excepciones).
- La operación ROLLBACK señala que la transacción no terminó con éxito, establece un punto de sincronización, cierra todos los cursores abiertos y libera todos los bloqueos (con algunas excepciones).

Recursos de bloqueo explícitos. DB2 maneja varios tipos distintos de bloqueo. Los usuarios de DB2 se ocupan principalmente de los bloqueos exclusivo (X) y compartido (S). Estudiaremos las diferentes clases de bloqueo y sus aplicaciones en el capítulo 18. Además del mecanismo de bloqueo interno, DB2 cuenta con algunos recursos de bloqueo explícitos. Una transacción puede emitir el siguiente enunciado:

```
LOCK TABLE <nombre-de-tabla> IN <tipo-de-modo> MODE;
```

El tipo-de-modo puede ser EXCLUSIVE (exclusivo) o SHARE (compartido). El nombre-de-tabla debe ser una tabla base. Un bloqueo exclusivo permite que la transacción bloquee por completo la tabla; el bloqueo se liberará cuando el programa (y no la transacción) termine. En cambio, el bloqueo compartido permite a otras transacciones adquirir un bloqueo compartido de manera concurrente sobre la misma tabla o una parte de la tabla. Mientras no se liberen todos los bloqueos compartidos, no será posible adquirir un bloqueo exclusivo sobre la tabla o una parte de ella.

Se proporciona este recurso con el fin de aumentar la eficiencia de las transacciones que necesitan procesar una sola tabla grande (por ejemplo, producir una lista de 10 000 empleados a partir de la tabla de empleados), sin incurrir en el costo extra de emitir y liberar bloqueos individuales a nivel de registro.

Si SQLCODE devuelve un valor negativo después de una operación de SQL que solicita un bloqueo significa que hay un bloqueo mortal. Para romper el bloqueo mortal, el gestor de transacciones escoge una de las transacciones bloqueadas como *víctima* y la hace revertir automáticamente, o le pide que ella misma se revierta. Esta transacción libera todos los bloqueos y permite a otra transacción continuar.

9.6.3 *SQL dinámico*

El recurso de SQL dinámico está diseñado exclusivamente para el manejo de aplicaciones en línea. En algunas aplicaciones caracterizadas por un alto grado de variabilidad, en vez de escribir una consulta en SQL específica para todas las posibles condiciones, puede ser mucho más conveniente que el diseñador construya partes de la consulta SQL dinámicamente (en el momento de la ejecución) y luego las enlace y ejecute dinámicamente. Este proceso tiene lugar cuando se introducen interactivamente instrucciones de SQL a través de DB2I o QMF. Sin embargo, consideraremos la forma de construir dinámicamente instrucciones de SQL *incorporado*. Sin entrar en detalles de la sintaxis, esbozaremos la naturaleza de este recurso.

Se define una cadena de caracteres en el lenguaje anfitrión con un cierto contenido inicial:

```
DCL CADCONS CHAR (256) VARYING INITIAL
'DELETE FROM EMPLEADO WHERE CONDICIÓN'.
```

Se declara una variable de SQL (en este caso VARBLESQL) para contener la consulta SQL durante la ejecución:

```
EXEC SQL DECLARE VARBLESQL STATEMENT;
```

CADCONS se modifica según sea apropiado cambiando, digamos, la CONDICIÓN en la parte WHERE de la consulta desde la terminal. La siguiente orden PREPARE hará que se precompile CADCONS, se enlace, se convierta en código objeto y se almacene en VARBLESQL:

```
EXEC SQL PREPARE VARBLESQL FROM :CADCONS;
```

Por último, la orden EXECUTE ejecuta realmente este código compilado:

```
EXEC SQL EXECUTE VARBLESQL;
```

Cabe señalar que PREPARE acepta todas las diferentes instrucciones de SQL, excepto EXEC SQL. Las instrucciones por preparar no pueden contener referencias a variables anfitrión, pero sí pueden contener parámetros denotados por signos de interrogación. Los valores de los parámetros se proporcionan en el momento de la ejecución mediante variables del programa. Por ejemplo, supongamos que la condición WHERE de CADCONS en nuestro ejemplo se reemplazara por "SALARIO > ? AND SALARIO < ?"; entonces

```
EXEC SQL EXECUTE VARBLESQL USING :LÍM_INF, :LÍM_SUP
```

sustituirá los dos signos de interrogación por los valores de las *variables de programa* LÍM_INF y LÍM_SUP.

Este análisis atañe las instrucciones de SQL que no devuelven datos al programa. Cuando es preciso obtener valores de datos a través de una instrucción SELECT generada dinámicamente, el programa casi nunca conoce las variables por anticipado. Así pues, la información se proporciona dinámicamente con otra instrucción de SQL dinámico, DESCRIBE (describir). La descripción de los resultados esperados se devuelve en un área llamada SQL *Descriptor Area* (SQLDA). A tales variables se les asigna memoria empleando el lenguaje de programación anfitrión. Por último, el resultado se obtiene fila por fila con operaciones de cursores. También es posible actualizar los resultados con la opción CURRENT (actual).

9.6.4 *Características para mejorar el rendimiento*

En esta sección mencionaremos algunos de los recursos con que cuenta la versión 2.3 de DB2 para mejorar el rendimiento de las aplicaciones. La lista que sigue, claro está, no es exhaustiva:

- Paquetes: Un programa fuente P1, después de la precompilación, puede convertirse en el módulo de solicitud de base de datos correspondiente, digamos DBRM P1 (véase la Fig. 9.2). Dos planes de aplicación distintos pueden usar el mismo DBRM P1. En tal caso, es posible definir un paquete para P1 e incluirlo en ambos planes. Si cambian los caminos de acceso que se usarán para DBRM P1, el paquete puede someterse otra vez al proceso de enlace, que se ejecutará *sólo una vez* con P1. Al cabo, lo que se ejecuta es un plan de aplicación y no el paquete, pero la noción de paquete intermedio evita que se dupliquen las actividades de enlace en los DBRM que se usan en muchos planes de aplicación.

- La cláusula OPTIMIZE for *n* ROWS (optimizar para *n* filas): Ésta puede añadirse a cualquier instrucción de SQL. Su propósito es suplantar la estimación calculada del número de filas que va a tener el resultado de una consulta. Al escoger un valor bajo de *n*, los usuarios pueden evitar el almacenamiento intermedio del resultado (llamado "preobtención de listas").

- La característica Index Lookaside (buscar al lado en el índice): Ésta es útil cuando los datos afectados por una consulta —en particular una actualización o una reunión— implican claves muy próximas entre sí. En vez de causar un examen repetitivo del índice, esta característica obliga al sistema a examinar las páginas hoja adyacentes del índice o el intervalo de páginas descendientes de un nodo intermedio del índice, antes de volver a la raíz para un examen descendente.

- Característica "SLOW CLOSE" (cierre lento): Permite que todos los conjuntos de datos (entre ellos los de tabla o de índice) permanezcan abiertos incluso después de que el usuario haya emitido la orden VSAM CLOSE, hasta que el número de conjuntos de datos abiertos exceda un cierto parámetro.

- Una característica relacionada con CICS para los cursores: En el ejemplo E2 de la sección 7.7 mostramos cómo debe abrirse un cursor cuando sea necesario ejecutar una consulta correspondiente de SQL incorporado en un programa de aplicación. Si se utiliza repetidamente el mismo cursor, en vez de emitir múltiples órdenes OPEN podemos realizar múltiples operaciones FETCH.

- Reuniones híbridas: Se han implementado nuevos algoritmos de reunión que combinan las características de una reunión de ciclo anidado y de una reunión de ordenación y combinación (analizadas en la Sec. 16.3). También permiten reducir el número de tablas temporales gracias al empleo de una lista de valores de clave coincidentes, lo cual corresponde a una técnica de reunión llamada "semirreunión".

Entre las características introducidas en las versiones 2.3 y 3 de DB2 tenemos las siguientes:

- Reservas de almacenamiento intermedio: En DB2 el usuario puede almacenar datos temporalmente en reservas de almacenamiento intermedio, con lo cual los datos están disponibles sin procesamiento de E/S. Sólo cuando se modifican estos datos se escriben en el disco físico. Podemos almacenar hasta 1.6 gigabytes de datos en estas reservas, denominadas **reservas de almacenamiento intermedio virtual**. La versión 3 de DB2 permite respaldar estas reservas con 8 gigabytes adicionales de almacenamiento expandido de acceso rápido. El almacenamiento expandido viene en unidades conocidas como *hiperespacios* (por ser espacios de alto rendimiento: *high-performance*), bloques de dos gigabytes que se construyen dinámicamente cuando se requieren para los datos almacenados en hiperreservas. Las *hiperreservas* (reservas de alto rendimiento) son extensiones de las reservas de almacenamiento intermedio virtual; los datos

de una reserva que no se utilizan con mucha frecuencia se pasan a su hiperreserva sin procesamiento de E/S.

- Independencia de particiones: En la versión 3 de DB2 una aplicación puede trabajar con una partición de un espacio de tablas o de índices sin bloquear las demás particiones; así, las particiones son independientes entre sí. Es posible aumentar la disponibilidad de los datos aprovechando la independencia de las particiones de dos maneras:
  - Realizar el mantenimiento en las particiones, no en espacios de tablas y de índices completos. Por ejemplo, recuperar, reorganizar o cargar una partición, y dejar la otra disponible para procesarla con utilerías o SQL.
  - Usar las órdenes START DATABASE (iniciar base de datos) y STOP DATABASE (detener base de datos) con particiones, no con espacios de tablas completos. Esto deja libres las demás particiones para procesarlas con utilerías o SQL.

9.7 Resumen

En este capítulo nos ocupamos de los diversos productos comerciales denominados sistemas de gestión de bases de datos relacionales (SGBDR), y examinamos a grandes rasgos las características de un producto muy importante: DB2 de IBM. El sistema DB2 debe su origen a un prototipo de investigación llamado System R de IBM. Existen otros SGBDR similares a él en la familia de DB2.

Presentamos la organización modular básica de DB2. Nuestro propósito no fue tanto ofrecer una descripción exhaustiva de estos módulos o de las características de DB2, como mostrar al lector un conjunto representativo de características internas y los detalles de organización de un SGBDR comercial. Señalamos que DB2 maneja básicamente el lenguaje SQL descrito en el capítulo 7, con detalles adicionales sobre la especificación y la imposición de la integridad referencial. A fin de ofrecer una descripción completa, también comentamos las características del procesamiento de transacciones y de la seguridad de DB2, temas cuyos conceptos se detallarán en los capítulos 17 a 20. Por último, señalamos algunas de las características introducidas en las versiones actuales de DB2: la 2.3 y la 3.

Apéndice del capítulo 9

E. F. Codd, el creador del modelo de datos relacional, publicó un artículo en dos partes en *Computerworld* (Codd 1985) que lista 12 reglas† para determinar si un SGBD es relacional o no, y hasta qué grado lo es [véase también Codd (1986)]. Las presentamos aquí porque proveen un criterio de medición muy útil para evaluar los sistemas relacionales. Codd también menciona que, según estas reglas, todavía no disponemos de un sistema completamente relacional. En particular, es difícil satisfacer las reglas 6, 9, 10, 11 y 12.

†Las reglas del apéndice se derivan de los dos artículos de Ted Codd que aparecieron en *Computerworld*: "Is your DBMS really relational?" (14 de octubre de 1985) y "Does your DBMS run by the rules?" (21 de octubre de 1985), derechos reservados © 1988 por CW Publishing, Inc.

Regla 1: Regla de información

Toda la información de una base de datos relacional se representa explícitamente en el nivel lógico de una y sólo una forma: mediante valores contenidos en tablas.

Regla 2: Regla de acceso garantizado

Se garantiza el acceso a todos y cada uno de los datos (valores atómicos) de una base de datos relacional a través de una combinación de nombre de tabla, valor de clave primaria y nombre de columna.

Regla 3: Tratamiento sistemático de valores nulos

El SGBD totalmente relacional maneja los valores nulos (distintos de la cadena de caracteres vacía o de una cadena de espacios en blanco, y distintos de cero y de cualquier otro número) para representar en forma sistemática cualquier información faltante, independiente del tipo de los datos.

Regla 4: Catálogo dinámico en línea basado en el modelo relacional

La descripción de la base de datos se representa en el nivel lógico de la misma manera que los datos ordinarios, de modo que los usuarios autorizados puedan aplicar el mismo lenguaje relacional para consultarla que el que aplican a los datos normales.

Regla 5: Regla de sublenguaje de datos completo

Un sistema relacional puede manejar varios lenguajes y diversos modos de uso terminal (por ejemplo, el modo de llenar los espacios en blanco). No obstante, debe haber por lo menos un lenguaje cuyos enunciados sean expresables, en alguna sintaxis bien definida, como cadenas de caracteres, y que pueda manejar todos los elementos siguientes: definición de datos, definición de vistas, manipulación de datos (interactiva y por programa), restricciones de integridad y cotas de transacciones (inicio, confirmación y reversión).

Regla 6: Regla de actualización de vistas

Todas las vistas que en teoría sean actualizables las podrá actualizar el sistema.

Regla 7: Inserción, modificación y eliminación de alto nivel

La capacidad de manejar una relación base o una relación derivada como un solo operando no sólo se aplica a la obtención de datos, sino también a su inserción, modificación y eliminación.

Regla 8: Independencia física de los datos

Los programas de aplicación y las actividades de terminal no serán afectados en su lógica cuando se haga cualquier cambio en la representación en almacenamiento o en los métodos de acceso.

Regla 9: Independencia lógica de los datos

Los programas de aplicación y las actividades de terminal no serán afectados en su lógica cuando se haga cualquier cambio en las tablas base que conservan la información, cuando teóricamente sea posible no afectar dichos programas y actividades.

Regla 10: Independencia de integridad

Las restricciones de integridad específicas para una base de datos relacional dada deben poderse definir en el sublenguaje de datos relacional y almacenarse en el catálogo, no en los programas de aplicación.

Se debe apoyar como mínimo estas dos restricciones de integridad:

1. Integridad de entidades: Ningún componente de una clave primaria puede tener un valor nulo.
2. Integridad referencial: Para cada valor distinto, no nulo, de clave externa en una base de datos relacional, debe existir un valor de clave primaria coincidente que pertenezca al mismo dominio.

Regla 11: Independencia de distribución

Un SGBD relacional tiene independencia de distribución; esto es, los usuarios no necesitan saber si una base de datos está distribuida o no.

Regla 12: Regla de no subversión

Si un sistema relacional tiene un lenguaje de bajo nivel (de un registro a la vez), ese lenguaje no puede usarse para subvertir o pasar por alto las reglas o restricciones de integridad expresadas en el lenguaje relacional de alto nivel (de varios registros a la vez).

Existe una cláusula adicional a estas 12 reglas, conocida como **regla cero**: "Cualquier sistema que califique como sistema de gestión de bases de datos relacionales debe ser capaz de manejar los datos enteramente a través de sus capacidades relacionales."

Según las reglas anteriores, todavía no contamos con ningún SGBD que sea totalmente relacional.

Bibliografía selecta

Hay varios libros dedicados a la descripción del sistema DB2, entre ellos: Date & White (1988), Martin, Chapman & Leben (1989), y Wiorkowski & Kull (1992). Chamberlin *et al.* (1981) ofrece una historia de System R, el precursor del sistema DB2, y del SQL/Data System. Blasgen *et al.* (1981) presenta un panorama de la arquitectura de System R. Codd (1990) describe el modelo relacional, versión 2, donde señala 18 categorías de características del modelo relacional y de su implementación en los SGBDR. El sistema DB2 se describe en varios manuales, incluidos los siguientes:

GC26-4886	Información general
SC26-4888	Guía de administración
SC26-4889	Guía de programación de aplicaciones y SQL
SC26-4891	Referencia de órdenes y utilerías
LY27-9603	Guía y referencia de diagnósticos
GC26-4887	Especificaciones de programas con licencia
GC26-4894	Índice maestro
SC26-4892	Mensajes y códigos
SX26-3801	Resumen de referencia
SC26-4890	Referencia de SQL
SC26-3077	Uso de órdenes de gestión de datos distribuidos

Entre otros interesantes manuales para la biblioteca sobre DBL están los siguientes:

GC26-4341	SAA: Generalidades
SC26-4650	Planificación para bases de datos relacionales distribuidas
GH24-5065	Conceptos y recursos de SQL/DS
SO4G-1022	Guía de programación y referencia de ES OS/2 Database Manager

Los recursos avanzados de DB2 se analizan en Lucyk (1993). Chang *et al.* (1988) ofrece un panorama del OS/2 Database Manager. La arquitectura de SAA se describe en IBM (1992). Mohan (1993) presenta un panorama sobre los productos de SGBDR de IBM. Hay varias publicaciones periódicas dedicadas al sistema DB2. Una de las más técnicas es la revista mensual *DB2 Journal*.

CAPÍTULO 10

El modelo de datos de red y el sistema IDMS

En los capítulos 6 a 9 estudiamos el modelo relacional de los datos, sus lenguajes y un SGBD relacional. Ahora veremos el modelo de red, que, junto con el modelo jerárquico, fue un modelo de datos importante para implementar un gran número de SGBD comerciales. Trataremos el modelo jerárquico en el siguiente capítulo. Desde el punto de vista histórico, las estructuras y construcciones del lenguaje para el modelo de red fueron definidas por el comité CODASYL (Conference on Data Systems Languages: Conferencia sobre lenguajes para sistemas de datos), por lo que suele denominársele modelo de red CODASYL. En fecha más reciente, el ANSI (American National Standards Institute: Instituto nacional estadounidense de estándares) publicó una recomendación para el establecimiento de una norma para el lenguaje de definición de red (NDL: *network definition languaje*) [ANSI 1984].

El modelo y el lenguaje de red originales se dieron a conocer en el informe de 1971 del Data Base Task Group (Grupo de trabajo sobre bases de datos) de CODASYL [DBTG 1971]; esto es lo que se conoce como modelo DBTG. En 1978 y 1981 se incorporaron conceptos más recientes en informes enmendados. En este capítulo, más que concentrarnos en los detalles de un informe CODASYL en particular, explicaremos los conceptos generales en que se sustentan las bases de datos tipo red y emplearemos el término **modelo de red** en vez de modelo CODASYL o modelo DBTG. Estudiaremos los conceptos del modelo de red independientemente de los modelos de datos de entidad-vínculo, relacional o jerárquico. En la sección 10.4 mostraremos cómo se diseña un esquema del modelo de red, a partir del modelo ER.

Cuando presentemos las órdenes para un lenguaje de base de datos de red nuestro lenguaje anfitrión será PASCAL, a fin de mantener la congruencia con el resto del libro. El informe CODASYL/DBTG original usó COBOL como lenguaje anfitrión. Sea cual fuere el lenguaje de programación anfitrión, las órdenes básicas para la manipulación de bases de datos del modelo de red siguen siendo las mismas.

En la sección 10.1 veremos los tipos de registros y los tipos de conjuntos, que son las dos construcciones de estructuración de datos más importantes para el modelo de red. En la sección 10.2 estudiaremos las restricciones del modelo de red, y en la sección 10.3 presentaremos un lenguaje de definición de datos (DDL) para dicho modelo. En la sección 10.4 mostraremos cómo se diseña un esquema de red estableciendo una transformación de un esquema ER conceptual al modelo de red. En la sección 10.5 presentaremos un lenguaje de manipulación de datos para las bases de datos de red, que es un lenguaje de registro por registro. Estos lenguajes contrastan con los lenguajes relacionales de alto nivel que vimos en los capítulos 6 a 9, que especifican la obtención de un conjunto de registros en una sola orden. Tradicionalmente, los modelos de red y jerárquico se asocian a lenguajes de bajo nivel, de registro por registro. La sección 10.6 ofrece un panorama del SGBD comercial de red IDMS.

Podrían omitirse algunas de las secciones 10.4 a 10.6, o todas, si no se requiere una exposición detallada del modelo de red.

10.1 Estructuras de una base de datos de red

Hay dos estructuras de datos básicas en el modelo de red: registros y conjuntos. Hablaremos de los registros y los tipos de registros en la sección 10.1.1. En la sección 10.1.2 presentaremos los conjuntos y sus propiedades básicas. En la sección 10.1.3 veremos algunos tipos especiales de conjuntos. En la sección 10.1.4 mostraremos cómo se representan e implementan los conjuntos. Por último, en la sección 10.1.5 mostraremos cómo se representan los vínculos 1:1 y M:*N* en el modelo de red.

10.1.1 *Registros, tipos de registros y elementos de información*

Los datos se almacenan en **registros**; cada registro consiste en un grupo de valores de datos relacionados entre sí. Los registros se clasifican en **tipos de registros**, cada uno de los cuales describe la estructura de un grupo de registros que almacenan el mismo tipo de información. Damos un nombre a cada tipo de registros, y también damos un nombre y un formato (tipo de datos) a cada **elemento de información** (o atributo) del tipo de registros. La figura 10.1 muestra un tipo de registros ESTUDIANTE con los elementos de información NOMBRE, NSS, DIRECCIÓN, DEPTOCARRERA y FECHANACIM. En la figura también se muestra el **formato** (o **tipo de datos**) de cada elemento de información.

Con el modelo de red es posible definir elementos de información complejos. Un **vector** es un elemento de información que puede tener múltiples valores en un solo registro.[†] Un **grupo repetitivo** permite incluir un conjunto de valores compuestos para un elemento de información en un solo registro.[††] Por ejemplo, si queremos incluir la boleta de notas de cada estudiante dentro de cada registro de estudiante, podemos definir un grupo repetitivo BOLETA para dicho registro; BOLETA consta de los cuatro elementos de información AÑO, CURSO, SEMESTRE y NOTAS, como se muestra en la figura 10.2. El grupo repetitivo no es esencial para la capacidad de modelado del modelo de red, ya que podemos representar la misma situación con dos tipos de registros y un tipo de conjuntos (véase la Sec. 10.1.2). La anidación de los grupos repetitivos puede llegar a varios niveles de profundidad.

[†]Esto corresponde a un atributo multivaluado *simple* en la terminología del capítulo 3.
[††]Esto corresponde a un atributo multivaluado *compuesto* en la terminología del capítulo 3.

ESTUDIANTE				
NOMBRE	NSS	DIRECCIÓN	DEPTOCARRERA	FECHANACIM

nombre de elemento de información	formato
NOMBRE	CHARACTER 30
NSS	CHARACTER 9
DIRECCIÓN	CHARACTER 40
DEPTOCARRERA	CHARACTER 10
FECHANACIM	CHARACTER 9

Figura 10.1 Tipo de registros ESTUDIANTE.

Todos los tipos de elementos de información que acabamos de mencionar se denominan **elementos de información reales**, porque sus valores se almacenan realmente en los registros. También es posible definir **elementos de información virtuales** (o **derivados**), cuyos valores no se almacenan realmente en los registros; en vez de ello, se derivan de los elementos de información reales mediante algún procedimiento definido específicamente para tal fin. Por ejemplo, podemos declarar un elemento de información virtual EDAD para el tipo de registros de la figura 10.1 y escribir un procedimiento que calcule el valor de EDAD a partir del valor del elemento de información real FECHANACIM de cada registro.

Una aplicación de base de datos común tiene muchos tipos de registros: de unos cuantos a varios centenares. Para representar los vínculos entre los registros, el modelo de red proporciona la construcción de modelado llamada *tipo de conjuntos*, que veremos a continuación.

10.1.2 *Tipos de conjuntos y sus propiedades básicas*

Un **tipo de conjuntos** es una descripción de un vínculo 1:*N* entre dos tipos de registros. La figura 10.3 muestra cómo se representa un tipo de conjuntos en un diagrama mediante una flecha. Este tipo de representación diagramática se llama **diagrama de Bachman**. Cada definición de tipo de conjuntos consta de tres elementos básicos:

- Un nombre para el tipo de conjuntos.
- Un tipo de registros propietario.
- Un tipo de registros miembro.

ESTUDIANTE					
NOMBRE	• • •	BOLETA			
		AÑO	CURSO	SEMESTRE	NOTAS
Silva	• • •	1984	CICO3320	Otoño	A
		1984	CICO3340	Otoño	A
		1984	MATE312	Otoño	B
		1985	CICO4310	Primavera	C
		1985	CICO4330	Primavera	B

Figura 10.2 Grupo repetitivo BOLETA.

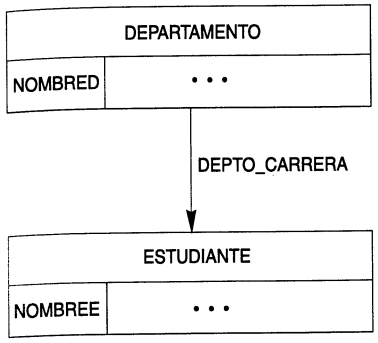


Figura 10.3 El tipo de conjuntos DEPTO_CARRERA (que es MANUAL OPTIONAL).

El tipo de conjuntos de la figura 10.3 se llama DEPTO_CARRERA, DEPARTAMENTO es el tipo de registros **propietario** y ESTUDIANTE es el tipo de registros **miembro**. Esto representa el vínculo 1:*N* entre los departamentos académicos y los estudiantes que cursan una carrera en esos departamentos. En la base de datos habrá muchas **ocurrencias de conjunto** (o **ejemplares de conjunto**) que corresponderán a un tipo de conjuntos. Cada ejemplar relaciona un registro del tipo de registros propietario —un registro DEPARTAMENTO en nuestro ejemplo— con el conjunto de registros del tipo de registros miembro relacionado con él: el conjunto de registros ESTUDIANTE de los estudiantes que cursan una carrera en ese departamento. Así, cada ocurrencia de conjunto se compone de:

- un registro propietario del tipo de registros propietario, y
- varios registros miembro relacionados (cero o más) del tipo de registros miembro.

Un registro del tipo de registros miembro *no puede existir en más de una ocurrencia de conjunto* de un tipo de conjuntos particular. Esto mantiene la restricción de que un tipo de conjuntos representa un vínculo 1:*N*. En nuestro ejemplo, un registro ESTUDIANTE puede estar relacionado cuando más con un DEPARTAMENTO de carrera y, por tanto, es miembro de cuando más una ocurrencia de conjunto del tipo de conjuntos DEPTO_CARRERA.

Una ocurrencia de conjuntos puede identificarse por el *registro propietario* o bien por *cualquiera de los registros miembro*. La figura 10.4 muestra cuatro ocurrencias (ejemplares) de conjunto del tipo de conjuntos DEPTO_CARRERA. Observe que cada ejemplar de conjunto *debe* tener un registro propietario pero puede tener cualquier número de registros miembro (**cero** o más). Por esto, casi siempre hacemos referencia a un ejemplar de conjunto por su registro propietario. Podríamos referirnos a los cuatro ejemplares de la figura 10.4 como los conjuntos 'Ciencias de la computación', 'Matemáticas', 'Física' y 'Geología'. Se acostumbra usar una representación distinta de los ejemplares de conjunto (Fig. 10.5), en la que los registros del ejemplar se muestran enlazados con apuntadores, pues esto corresponde a una técnica muy común para implementar conjuntos.

En el modelo de red, un ejemplar de conjunto *no es idéntico* al concepto matemático de conjunto. Sus principales diferencias son dos:

- El ejemplar de conjunto tiene un *elemento distinguido* —el registro propietario— pero en un conjunto matemático no hay distinción alguna entre los elementos.

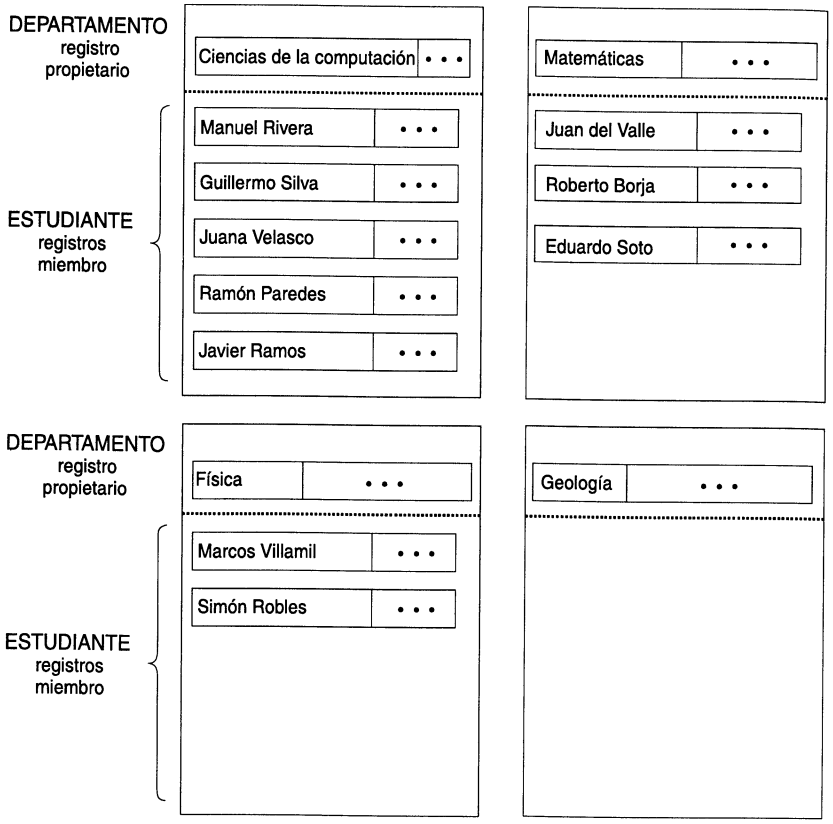


Figura 10.4 Cuatro ejemplares de conjunto del tipo de conjuntos DEPTO_CARRERA.

- En el modelo de red los registros miembro de un ejemplar de conjunto están *ordenados*, mientras que en un conjunto matemático el orden de los elementos carece de importancia. Así, podemos hablar del primero, segundo, *i*-ésimo y último registros miembro de un ejemplar de conjunto. La figura 10.5 muestra otra representación "enlazada" de un ejemplar del conjunto DEPTO_CARRERA. En ella, el registro de 'Manuel Rivera' es el primer registro (miembro) ESTUDIANTE del conjunto 'Ciencias de la computación', y el de 'Javier Ramos' es el último. A los conjuntos del modelo de red también se les llama **conjuntos acoplados al propietario** o **co-conjuntos**, a fin de distinguirlos de los conjuntos matemáticos.

10.1.3 *Tipos especiales de conjuntos*

En el modelo de red CODASYL hay dos tipos especiales de conjuntos: los conjuntos propiedad del sistema y los conjuntos multimiembro. Un tercer tipo, el llamado conjunto recursivo, no estaba permitido en el informe CODASYL original. A continuación analizaremos estos tres tipos de conjuntos especiales.

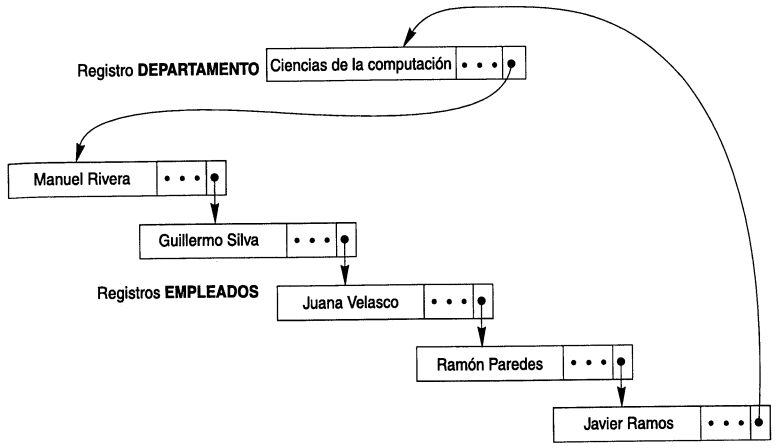


Figura 10.5 Representación alternativa de un ejemplar de conjunto.

Conjuntos propiedad del sistema (singulares). Un **conjunto propiedad del sistema** es un conjunto sin tipo de registros propietario; en este caso, el sistema† es el propietario. Podemos considerar al sistema como un tipo de registro propietario "virtuales" especial en el que sólo hay una ocurrencia de registro. Los conjuntos propiedad del sistema tienen dos funciones principales en el modelo de red:

- Proveen *puntos de entrada* a la base de datos a través de los registros del tipo de registros miembro especificado. El procesamiento puede comenzar con la obtención de acceso a los miembros de ese tipo de registros, llegando después a los registros relacionados a través de otros conjuntos.
- Pueden servir para *ordenar* los registros de un tipo de registros dado mediante las especificaciones de ordenamiento del conjunto. Si un usuario especifica varios conjuntos propiedad del sistema para el mismo tipo de registros, puede tener acceso a sus registros en diferentes órdenes.

Un conjunto propiedad del sistema permite procesar los registros de un tipo de registros con las operaciones normales de conjuntos que veremos en la sección 10.5.3. Este tipo de conjuntos se denomina conjunto **singular** porque sólo hay una ocurrencia de ese conjunto. En la figura 10.6(a) se muestra la representación diagramática del conjunto propiedad del sistema TODOS_DEPTOS, que permite tener acceso a los registros DEPARTAMENTO en orden según cierto campo —digamos, NOMBRE— con una especificación apropiada de ordenamiento de conjunto.

Conjuntos multimiembro. Los conjuntos **multimiembro** se utilizan en casos en los que los registros miembro de un conjunto pueden pertenecer a *más de un* tipo de registros. Casi ningún SGBD de red comercial los maneja. Los registros miembro en una ocurrencia de conjunto de un tipo de conjuntos multimiembro pueden contener registros de cualquier combinación de tipos de registros miembro. En la figura 10.6(b) se muestra un conjunto

†Con *sistema* nos referimos al software del SGBD.

multimiembro con tres tipos de registros miembro. La restricción de que cada registro miembro pueda aparecer en una ocurrencia de conjunto, como máximo, sigue siendo válida, a fin de imponer la naturaleza 1:*N* del vínculo.

Conjuntos recursivos. Un conjunto **recursivo** es un tipo de conjuntos en el que el mismo tipo de registros desempeña el papel tanto de propietario como de miembro. Un ejemplo de vínculo recursivo 1:*N* que se puede representar con un conjunto recursivo es el vínculo SUPERVISIÓN, que relaciona un empleado supervisor con la lista de empleados a los que supervisa directamente. En este vínculo, el tipo de registros EMPLEADO desempeña ambos papeles: el de tipo de registros propietario (el empleado supervisor) y el de tipo de registros miembro (los empleados supervisados).

En el modelo CODASYL original estaban prohibidos los conjuntos recursivos debido a la dificultad de procesarlos con el lenguaje de manipulación de datos (DML) de CODASYL. En el DML (véase la Sec. 10.5.2) se supone que un registro pertenece a una sola ocurrencia de conjunto de cada tipo de conjuntos. En el caso de los conjuntos recursivos, el mismo registro puede ser propietario de una ocurrencia de conjunto y miembro de otra, si ambas ocurrencias de conjunto son del mismo tipo de conjuntos. Por este problema, se acostumbra representar los conjuntos recursivos en el modelo de red con un tipo de registros **de enlace** (o **ficticio**) adicional. La misma técnica se usa para representar vínculos M:*N*, como veremos en la sección 10.1.5. La figura 10.6(c) muestra la representación del vínculo SUPERVISIÓN, empleando dos tipos de conjuntos y un tipo de registros de enlace. En la figura 10.6(c) el tipo de conjuntos SUPERVISOR en realidad es un vínculo 1:1; es decir, hay cuando más un tipo de registros SUPERVISIÓN miembro en cada ocurrencia del conjunto SUPERVISOR. Podemos considerar cada registro de enlace SUPERVISIÓN como la representación de un empleado *en el papel*

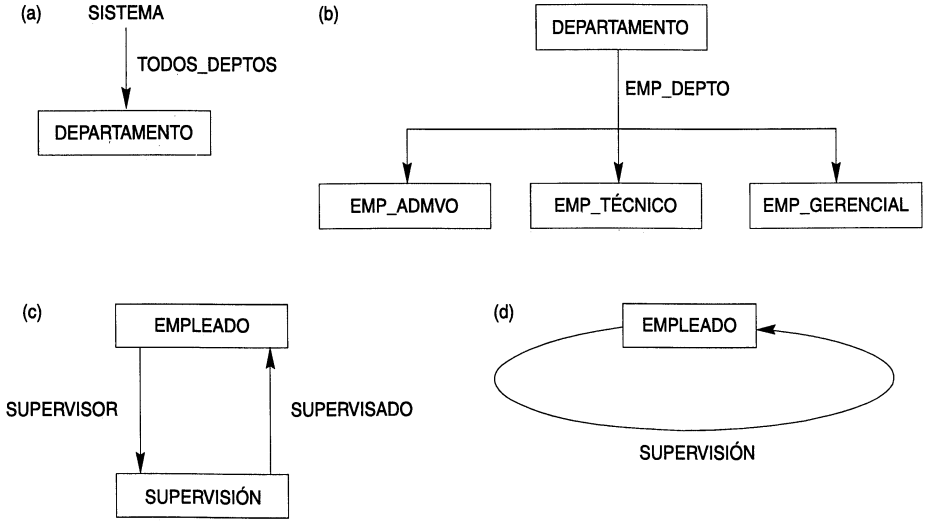


Figura 10.6 Tipos especiales de conjuntos. (a) Un conjunto singular (propiedad del sistema). (b) Un conjunto multimiembro. (c) El conjunto recursivo SUPERVISIÓN representado mediante un tipo de registros de enlace. (d) Representación prohibida de un conjunto recursivo.

de supervisor. La representación directa de un conjunto recursivo —por lo regular prohibida en el modelo de red— se muestra en la figura 10.6(d). Casi ninguna implementación de SGBD de red permite al mismo tipo de registros participar como propietario y como miembro en el mismo tipo de conjuntos.

10.1.4 *Representaciones almacenadas de ejemplares de conjuntos*

Los ejemplares de conjuntos suelen representarse como **anillos (listas enlazadas circulares)** que enlazan el registro propietario con todos los registros miembro del conjunto, como se muestra en la figura 10.5. Esto se conoce también como **cadena circular**. Esta representación anular es simétrica con respecto a todos los registros; por ello, para distinguir el registro propietario de los registros miembro, el SGBD cuenta con un campo especial, el llamado **campo de tipo**, que tiene un valor distinto (asignado por el SGBD) para cada tipo de registros. Al examinar el campo de tipo, el sistema puede saber si el registro es el propietario del ejemplar de conjunto o es uno de los registros miembro. El usuario no puede ver este campo de tipo; sólo el SGBD lo utiliza.

Además del campo de tipo, el SGBD asigna automáticamente a cada tipo de registros un **campo apuntador** por *cada tipo de conjuntos en el que participa como propietario o como miembro*. Podemos considerar que este apuntador está *rotulado* con el nombre del tipo de conjuntos correspondiente; así, el sistema mantiene internamente la correspondencia entre estos campos apuntadores y sus tipos de conjuntos. Es común llamar **SIGUIENTE** al apuntador de un registro miembro y **PRIMERO** al apuntador de un registro propietario, porque apuntan al siguiente y al primer registros miembro, respectivamente. En nuestro ejemplo de la figura 10.5, cada registro ESTUDIANTE tiene un apuntador SIGUIENTE al siguiente registro ESTUDIANTE dentro de la ocurrencia de conjunto. El apuntador SIGUIENTE del *último registro miembro* de una ocurrencia de conjunto apunta al registro propietario. Si un registro del tipo de registros miembro no participa en ningún ejemplar de conjunto, su apuntador SIGUIENTE tendrá un apuntador **nil** especial. Si una ocurrencia de conjunto tiene propietario pero no tiene registros miembro, el apuntador PRIMERO apuntará a ese registro propietario, aunque también puede ser **nil**.

La representación de conjuntos que hemos visto es un método para implementar los ejemplares de conjuntos. En general, un SGBD puede implementar los conjuntos de diversas maneras, pero la representación elegida deberá permitir al sistema realizar todas estas operaciones:

- Dado un registro propietario, encontrar todos los registros miembro de la ocurrencia de conjunto.
- Dado un registro propietario, encontrar el primero, *i*-ésimo o último registro miembro de la ocurrencia de conjunto. Si no existe ningún registro así, indicar ese hecho.
- Dado un registro miembro, encontrar el siguiente registro miembro (o el anterior) de la ocurrencia de conjunto. Si no existe ningún registro así, indicar ese hecho.
- Dado un registro miembro, encontrar el registro propietario de la ocurrencia de conjunto.

La representación de lista enlazada circular permite al sistema realizar todas estas operaciones con diversos grados de eficiencia. En general, un esquema de base de datos de red tiene muchos tipos de registros y de conjuntos, así que un tipo de registros puede participar como

propietario y miembro en un gran número de tipos de conjuntos. Por ejemplo, en el esquema de red que aparece más adelante en la figura 10.9, el tipo de registros EMPLEADO participa como propietario en cuatro tipos de conjuntos —DIRIGE, ES_SUPERVISOR, E_TRABAJAEN y DEPENDIENTES_DE— y participa como miembro en dos tipos de conjuntos: PERTENECE_A y SUPERVISADOS. En la representación de lista enlazada circular, se añaden seis campos apuntadores adicionales al tipo de registros EMPLEADO. Sin embargo, no hay confusión, porque el sistema rotula cada uno de los apuntadores, los cuales desempeñan el papel de apuntador PRIMERO o SIGUIENTE para un tipo de conjuntos específico.

Con otras representaciones de conjuntos es posible implementar con mayor eficiencia algunas de las operaciones de conjuntos que acabamos de ver. Aquí haremos una breve mención de cinco de ellas:

- Representación de lista circular con doble enlace: Además del apuntador SIGUIENTE de un tipo de registros miembro, un apuntador **PREVIO** apunta al registro miembro anterior de la ocurrencia de conjunto. El apuntador PREVIO del primer registro miembro puede apuntar al registro propietario.
- Representación de apuntador al propietario: Puede utilizarse en combinación ya sea con la representación de lista enlazada o con la de lista doblemente enlazada. Para cada tipo de conjuntos, se incluye un apuntador PROPIETARIO adicional en el tipo de registros miembro que apunta directamente al registro propietario del conjunto.
- Registros miembro contiguos: En vez de estar enlazados por apuntadores, los registros miembro se colocan en posiciones físicamente contiguas, casi siempre en seguida del registro propietario.
- Arreglos de apuntadores: Un arreglo de apuntadores se almacena junto con el registro propietario. El *i*-ésimo elemento del arreglo apunta al *i*-ésimo registro miembro de la ocurrencia de conjunto. Esto suele implementarse junto con el apuntador al propietario.
- Representación indizada: Se guarda un índice pequeño con el registro propietario *por cada ocurrencia de conjunto*. Cada entrada del índice contiene un valor de un campo clave de indización y un apuntador al registro miembro que tiene ese valor en dicho campo. El índice puede implementarse como lista enlazada encadenada mediante apuntadores SIGUIENTE y PREVIO (el sistema IDMS cuenta con esta opción; véase la Sec. 10.6).

Estas representaciones apoyan las operaciones del DML de red con diferentes grados de eficiencia. Idealmente, el programador no deberá ocuparse de cómo han de implementarse los conjuntos; sólo deberá confirmar que el SGBD los haya implementado correctamente. No obstante, en la práctica, el programador puede aprovechar la implementación específica de los conjuntos para escribir programas más eficientes. Con casi todos los sistemas el diseñador de la base de datos puede elegir entre varias opciones para implementar cada tipo de conjuntos con una instrucción MODE (modo) para especificar la representación elegida.

10.1.5 *Empleo de conjuntos para representar vínculos 1:1 y M:N*

Un tipo de conjuntos representa un vínculo 1:*N* entre dos tipos de registros. Esto significa que *un registro del tipo de registros miembro sólo puede aparecer en una ocurrencia del conjunto*. El SGBD impone automáticamente esta restricción en el modelo de red.

Si queremos representar un vínculo 1:1 entre dos tipos de registros con un tipo de conjuntos, debemos hacer que cada ocurrencia de conjunto sólo pueda tener *un registro miembro*. El modelo de red no cuenta con mecanismos para imponer automáticamente esta restricción, por lo que el *programador* debe comprobar que no se viole esta restricción cada vez que se inserta un registro miembro en una ocurrencia de conjunto.

Un vínculo M:*N* entre dos tipos de registros no puede representarse con un solo tipo de conjuntos. Por ejemplo, consideremos el vínculo TRABAJA_EN entre EMPLEADOs y PROYECTOs. Supongamos que un empleado puede trabajar en varios proyectos al mismo tiempo y que un proyecto casi siempre tiene varios empleados que trabajan en él. Si tratamos de representar esto con un tipo de conjuntos, ni el tipo de conjuntos de la figura 10.7(a) ni el de la figura 10.7(b) representarán correctamente el vínculo. La figura 10.7(a) impone la restricción incorrecta de que un registro PROYECTO esté relacionado con un solo registro EMPLEADO, mientras que la figura 10.7(b) impone la restricción incorrecta de que un registro EMPLEADO esté relacionado con un solo registro PROYECTO. Si usamos ambos tipos de conjuntos, E_P y P_E, simultáneamente, como en la figura 10.7(c), surge el problema de imponer la restricción de que E_P y P_E sean inversos mutuamente consistentes, además del problema de manejar atributos del vínculo.

El método correcto para representar un vínculo M:*N* en el modelo de red es utilizar dos tipos de conjuntos y un tipo de registros adicionales, como se aprecia en la figura 10.7(d). Este tipo de registros adicional —TRABAJA_EN, en nuestro ejemplo— se llama tipo de registros **de enlace** (o **ficticio**). Cada registro del tipo de registros TRABAJA_EN debe ser propiedad de un registro EMPLEADO a través del conjunto E_T y de un registro PROYECTO a través del conjunto P_T, y sirve para relacionar estos dos registros propietario. Esto se ilustra conceptualmente en la figura 10.7(e).

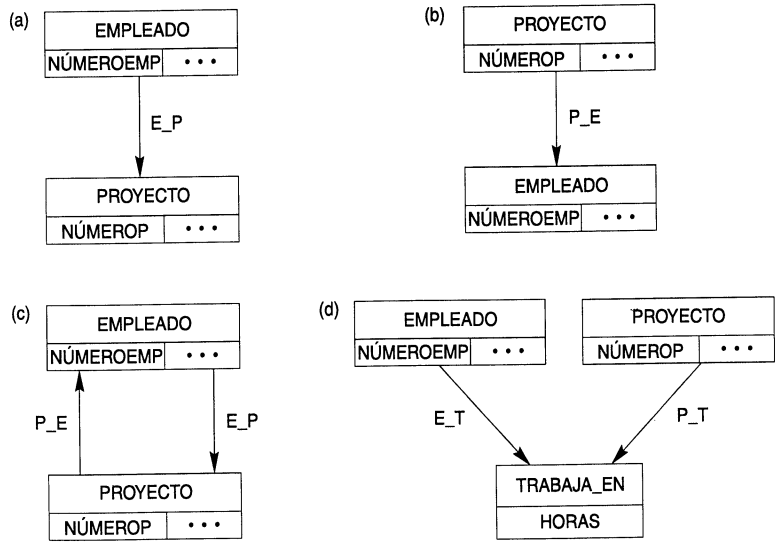


Figura 10.7 Representación de vínculos M:*N* mediante conjuntos. (a)-(c) Representaciones incorrectas de un vínculo M:N. (d) Representación correcta de un vínculo M:*N* empleando un tipo de registros de enlace (ficticio). (*continúa en la página siguiente*)

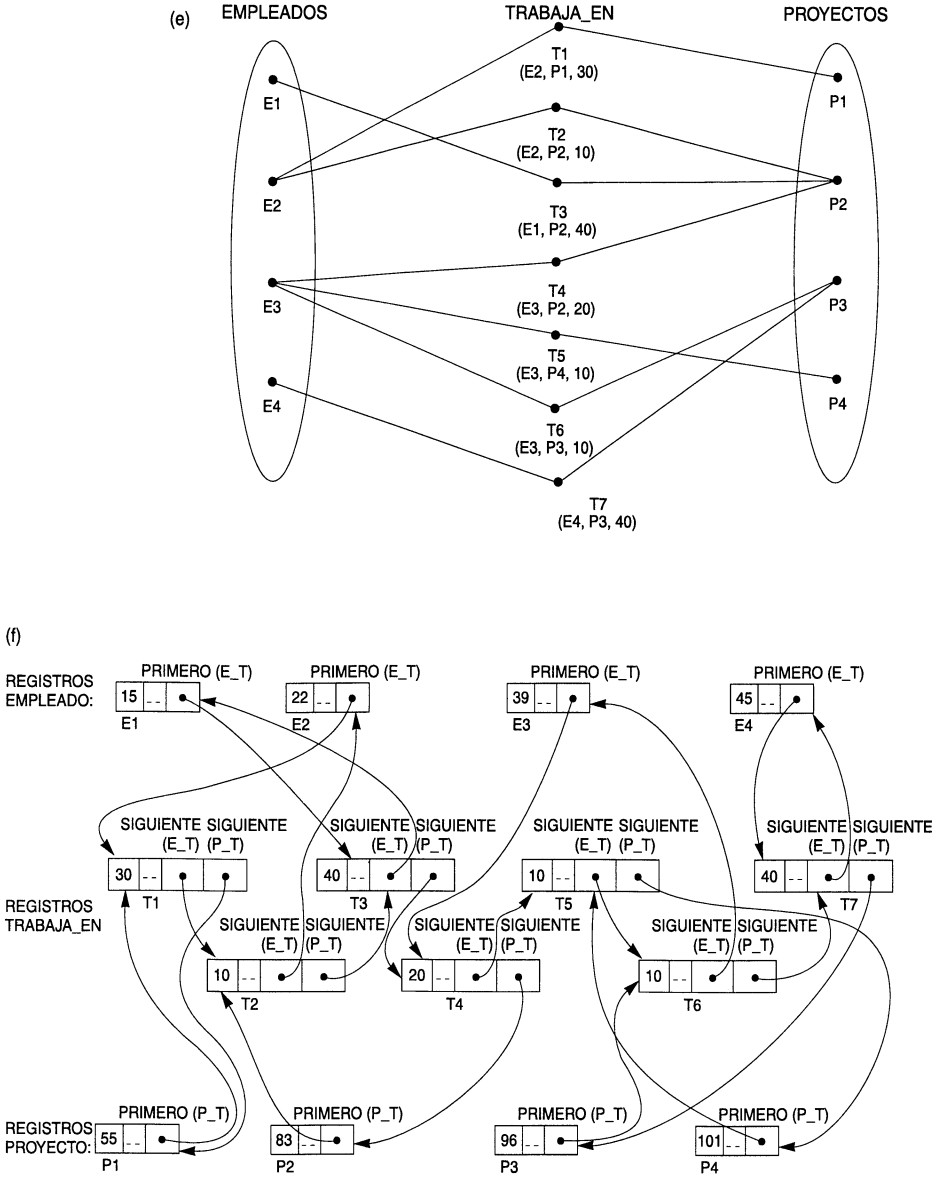


Figura 10.7 (*continuación*) (e) Representación de algunas ocurrencias de un vínculo M:*N* con "ocurrencias enlazadas". (f) Algunas ocurrencias de los tipos de conjuntos E_T y P_T y del tipo de registros de enlace TRABAJA_EN correspondientes a los ejemplares del vínculo M:*N* que se muestran en la figura 10.7(e).

La figura 10.7(f) muestra un ejemplo de ocurrencias individuales de registro y de conjunto en la representación de lista enlazada que corresponde al esquema de la figura 10.7(d). Cada registro del tipo de registros TRABAJA_EN tiene dos apuntadores SIGUIENTE: el rotulado SIGUIENTE(E_T) apunta al siguiente registro en un ejemplar del conjunto E_T, y el rotulado SIGUIENTE(P_T) apunta al siguiente registro en un ejemplar del conjunto P_T. Cada registro TRABAJA_EN relaciona sus dos registros propietario, y además contiene el número de horas que un empleado trabaja por semana en un proyecto. En la figura 10.7(e), que ilustra las mismas ocurrencias que la figura 10.7(f), se muestran los registros T individualmente, sin los apuntadores.

Si queremos encontrar todos los proyectos en los que trabaja un cierto empleado, comenzamos en el registro EMPLEADO y recorremos todos los registros TRABAJA_EN propiedad de ese empleado, empleando los apuntadores PRIMERO(E_T) y SIGUIENTE(E_T). En cada registro TRABAJA_EN de la ocurrencia de conjunto encontramos su registro PROYECTO propietario siguiendo los apuntadores SIGUIENTE(P_T) hasta hallar un registro de tipo PROYECTO. Por ejemplo, en el caso del registro EMPLEADO E2, seguimos el apuntador PRIMERO(E_T) de E2 que conduce a T1, el apuntador SIGUIENTE(E_T) de T1 que conduce a T2 y el apuntador SIGUIENTE(E_T) de T2 que conduce de vuelta a E2. Así, establecemos que T1 y T2 son los registros miembro de la ocurrencia de conjunto E_T propiedad de E2. Si seguimos el apuntador SIGUIENTE(P_T) de T1 llegamos a su propietario, P1, y si seguimos el apuntador SIGUIENTE(P_T) de T2 (a través de T3 y T4) llegamos a su propietario, P2. Adviértase que la existencia de apuntadores PROPIETARIO directos para el conjunto P_T en los registros TRABAJA_EN habría simplificado el proceso de identificar el registro PROYECTO propietario de cada registro TRABAJA_EN.

En forma similar, podemos encontrar todos los registros EMPLEADO relacionados con un PROYECTO en particular. En este caso la existencia de apuntadores PROPIETARIO para el conjunto E_T simplificaría el procesamiento. *El SGBD realiza automáticamente* todo este rastreo de apuntadores; el programador dispone de órdenes en DML para buscar directamente el propietario o el siguiente miembro, como veremos en la sección 10.5.3.

Cabe señalar que podríamos representar el vínculo M:*N* como en la figura 10.7(a) (o como en la 10.7(b)) si se nos permitiera duplicar registros PROYECTO (o EMPLEADO). En la figura 10.7(a) se repetiría un registro PROYECTO tantas veces como empleados trabajaran en el proyecto. Sin embargo, siempre que se actualiza la base de datos la duplicación de registros es problemática para mantener la consistencia entre los duplicados, y en general no es recomendable.

10.2 Restricciones en el modelo de red

En nuestra explicación del modelo de red, ya hemos hablado de restricciones "estructurales" que gobiernan la forma como están estructurados los tipos de registros y de conjuntos. En esta sección estudiaremos las restricciones "de comportamiento" que se aplican a los miembros de los conjuntos (o bien al comportamiento de dichos miembros) cuando se realizan operaciones de inserción, eliminación y actualización con esos conjuntos. Podemos especificar varias restricciones sobre la pertenencia a un conjunto, y éstas suelen dividirse en dos categorías principales, llamadas **opciones de inserción** y **opciones de retención** en la terminología CODASYL. Para determinar estas restricciones durante el diseño de la base de

datos hay que conocer cómo deberá *comportarse* un conjunto cuando se inserten registros miembro o cuando se eliminen registros miembro o propietario. Las restricciones se especifican al SGBD cuando se declara la estructura de la base de datos empleando el lenguaje de definición de datos (véase la Sec. 10.3). No todas las combinaciones de las restricciones son posibles. Primero examinaremos cada tipo de restricción y luego presentaremos las combinaciones permitidas.

10.2.1 *Opciones (restricciones) de inserción en conjuntos*

Las restricciones —u opciones, en la terminología CODASYL— de inserción sobre la pertenencia a conjuntos especifican lo que sucede cuando se inserta en la base de datos un registro nuevo que es de un tipo de registros miembro. Los registros se insertan con la orden STORE (almacenar, véase la Sec. 10.5.4). Hay dos opciones de inserción:

- AUTOMATIC: El nuevo registro miembro *se conecta automáticamente* a una ocurrencia de conjunto apropiada† cuando se inserta el registro.
- MANUAL: El nuevo registro no se conecta a ninguna ocurrencia de conjunto. Si el programador lo desea, puede conectar después explícitamente (*manualmente*) el registro a una ocurrencia de conjunto, mediante la orden CONNECT.

Por ejemplo, consideremos el tipo de conjuntos DEPTO_CARRERA de la figura 10.3. En esta situación podemos tener un registro ESTUDIANTE que no esté relacionado con ningún departamento a través del conjunto DEPTO_CARRERA (si el estudiante en cuestión no se ha decidido por una carrera). Por tanto, deberemos declarar la opción de inserción MANUAL, para que cada vez que se inserte un registro miembro ESTUDIANTE en la base de datos no se relacione automáticamente con un registro DEPARTAMENTO a través del conjunto DEPTO_CARRERA. Más adelante, el usuario de la base de datos puede insertar el registro "manualmente" en un ejemplar de conjunto cuando el estudiante se decida por una carrera. Esta inserción manual se logra con una operación de actualización llamada CONNECT, presentada al sistema de base de datos, como veremos en la sección 10.5.4.

La opción AUTOMATIC para la inserción en conjuntos se usa en situaciones en las que deseamos insertar un registro miembro en un ejemplar de conjunto automáticamente cuando almacenamos el registro en la base de datos. Debemos especificar un criterio para *designar el ejemplar de conjunto* al cual pertenecerá cada registro nuevo. A guisa de ejemplo, consideremos el tipo de conjuntos de la figura 10.8(a), que relaciona cada empleado con el conjunto de sus dependientes. Podemos declarar que sea AUTOMATIC el tipo de conjuntos DEPENDIENTES_EMP, con la condición de que un registro DEPENDIENTE nuevo con un valor de NSSEMP dado se inserte en el ejemplar de conjunto propiedad del registro EMPLEADO que tiene ese mismo valor de NSS. El SGBD localizará el registro EMPLEADO tal que EMPLEADO.NSS = DEPENDIENTE.NSSEMP y conectará automáticamente el nuevo registro DEPENDIENTE a ese ejemplar de conjunto. Observe que el campo NSS debe declararse de modo que no haya dos registros EMPLEADO con el mismo NSS; de lo contrario, la condición anterior identificará más de un ejemplar de conjunto. En la sección 10.3.2 veremos otros criterios para identificar y seleccionar automáticamente una ocurrencia de conjunto.

†La ocurrencia de conjunto apropiada se determina con una especificación que forma parte de la definición del tipo de conjuntos, la SET OCURRENCE SELECTION, que estudiaremos en la sección 10.3.2 como parte del DDL de red.

10.2.2 *Opciones (restricciones) de retención en conjuntos*

Las restricciones —u opciones, en la terminología CODASYL— de retención especifican si un registro de un tipo de registros miembro puede existir en la base de datos por sí solo o si siempre debe estar relacionado con un propietario como miembro de algún ejemplar de conjunto. Hay tres opciones de retención:

- OPTIONAL (opcional): Un registro miembro puede existir por sí solo *sin ser* miembro de ninguna ocurrencia del conjunto. Se le puede conectar y desconectar de las ocurrencias de conjunto a voluntad con las órdenes CONNECT y DISCONNECT del DML de red (véase la Sec. 10.5.4).
- MANDATORY (obligatoria): *Ningún* registro miembro puede existir por sí solo; *siempre* debe ser miembro de una ocurrencia de conjunto del tipo de conjuntos. Se le puede reconectar en una sola operación de una ocurrencia de conjunto a otra mediante la orden RECONNECT del DML de red (véase la Sec. 10.5.4).
- FIXED (fija): Al igual que en MANDATORY, *ningún* registro miembro puede existir por sí solo. Por añadidura, una vez insertado en una ocurrencia de conjunto, queda *fijo*; *no se le puede* reconectar a otra ocurrencia de conjunto.

Ahora ilustraremos las diferencias entre estas opciones con ejemplos de las situaciones en las que debe usarse cada opción. Primero, consideremos el tipo de conjuntos DEPTO_CARRERA de la figura 10.3. Previendo la situación en la que un registro ESTUDIANTE no esté relacionado con ningún departamento a través del conjunto DEPTO_CARRERA, declaramos

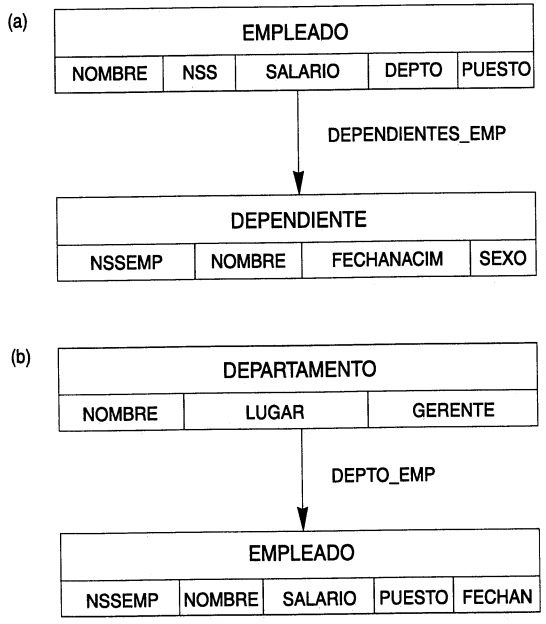


Figura 10.8 Diferentes opciones de conjuntos. (a) Tipo de conjuntos DEPENDIENTES_EMP AUTOMATIC FIXED. (b) Tipo de conjuntos DEPTO_EMP AUTOMATIC MANDATORY.

Tabla 10.1 Opciones de inserción y retención en conjuntos

Opción de inserción \ *Opción de retención*	OPTIONAL	MANDATORY	FIXED
MANUAL	El programa de aplicación se encarga de insertar el registro miembro en la ocurrencia de conjunto. Se puede CONECTAR, DESCONECTAR Y RECONECTAR.	No es muy útil.	No es muy útil.
AUTOMATIC	El SGBD inserta automáticamente el nuevo registro miembro en una ocurrencia de conjunto. Se puede CONECTAR, DESCONECTAR Y RECONECTAR.	El SGBD inserta automáticamente el nuevo registro miembro en una ocurrencia de conjunto. Se puede RECONECTAR un miembro a un propietario distinto.	El SGBD inserta automáticamente el nuevo registro miembro en una ocurrencia de conjunto. *No se puede* RECONECTAR un miembro a un propietario distinto.

que el conjunto es OPTIONAL. En la figura 10.8(a), DEPENDIENTES_EMP es un ejemplo de tipo de conjuntos FIXED, porque no es de esperarse que pasemos un dependiente de algún empleado a otro. Además, todo registro DEPENDIENTE debe estar relacionado con algún registro EMPLEADO en todo momento. En la figura 10.8(b) un conjunto MANDATORY DEPTO_EMP relaciona un empleado con el departamento al que pertenece. Aquí, cada empleado debe estar asignado a uno y sólo un departamento en todo momento; sin embargo, es posible reasignar un empleado de un departamento a otro.

En general, las opciones MANDATORY y FIXED se usan en situaciones en las que un registro miembro no debe existir en la base de datos sin estar relacionado con un propietario a través de alguna ocurrencia de conjunto. En el caso de FIXED, se impone el requerimiento adicional de nunca pasar un registro miembro de un ejemplar de conjunto a otro. Si utiliza la opción de inserción/retención apropiada, el DBA será capaz de especificar el comportamiento de un tipo de conjuntos en forma de restricción, y el sistema *automáticamente* hará que se cumpla.

10.2.3 *Combinaciones de opciones de inserción y retención*

No todas las combinaciones de opciones de inserción y retención son útiles. Por ejemplo, las opciones FIXED y MANDATORY implican que un registro miembro siempre debe estar relacionado con un propietario, así que deben usarse con la opción de inserción AUTOMATIC. Si bien cualquier combinación de estas opciones es técnicamente válida, por lo regular sólo tres combinaciones tienen sentido, y la mayor parte de las implementaciones del modelo de red sólo permiten estas tres combinaciones "razonables": AUTOMATIC-FIXED, AUTOMATIC-MANDATORY y MANUAL-OPTIONAL.† También podemos imaginar aplicaciones en las que un

†El informe CODASYL DBTG original no aplicaba estas restricciones a las posibles combinaciones de opciones.

conjunto AUTOMATIC-OPTIONAL podría ser útil; a saber, cuando el nuevo registro miembro se conecta automáticamente a un propietario si se especifica un propietario en particular, pero no se conecta a ningún ejemplar de conjunto en caso contrario. Estas combinaciones se resumen en la tabla 10.1.

10.2.4 *Opciones de ordenamiento de conjuntos*

Los registros miembro de un ejemplar de conjunto pueden estar ordenados de diversas maneras. El orden puede basarse en un campo de ordenamiento o provenir de la secuencia temporal de inserción de los registros miembro nuevos. Las opciones de ordenamiento disponibles se pueden resumir como sigue:

- Ordenadas según un campo de ordenamiento: Los valores de uno o más campos del registro miembro sirven para ordenar dichos registros dentro de *cada ocurrencia de conjunto* en orden ascendente o descendente. El sistema mantiene el orden cuando se conecta un nuevo registro miembro al ejemplar de conjunto insertando automáticamente el registro en su posición correcta.
- Orden por omisión del sistema: Un nuevo registro miembro se inserta en una posición arbitraria determinada por el sistema.
- Primero o último: Un nuevo registro miembro se convierte en el primero o en el último miembro de la ocurrencia de conjunto *en el momento en que se inserta*. Por tanto, esto equivale a tener los registros miembro de un ejemplar de conjunto almacenados en orden cronológico (o cronológico invertido).
- Siguiente o previo: El nuevo registro miembro se inserta después o antes del registro actual de la ocurrencia de conjunto. Esto se aclarará cuando analicemos los indicadores de actualidad (*currency indicators*) en la sección 10.5.1.

Las opciones deseadas en cuanto a inserción, retención u ordenamiento se especifican cuando se declara el tipo de conjuntos en el lenguaje de definición de datos. Los detalles de la declaración de tipos de registros y de conjuntos se analizarán en la sección 10.3 en relación con el lenguaje de definición de datos (DDL) del modelo de red.

10.3 Definición de datos en el modelo de red

Después de diseñar un esquema de base de datos de red, debemos declarar al SGBD todos los tipos de registros, tipos de conjuntos, definiciones de elementos de información y restricciones del esquema. Para ello, usamos el DDL de red. Cada SGBD de red tiene una sintaxis un tanto distinta y opciones con pequeñas diferencias en su DDL, de modo que en vez de presentar la sintaxis exacta del DDL del SGBD CODASYL nos concentraremos en entender los diferentes conceptos y opciones disponibles en casi todos los SGBD de red.

10.3.1 *Declaraciones de tipos de registros y de elementos de información*

Las declaraciones de DDL de red para los tipos de registros del esquema COMPAÑÍA que se muestra en la figura 10.9 aparecen en la figura 10.10(a). Cada tipo de registros recibe un nombre a través de la cláusula **RECORD NAME IS** (nombre de registro es). Se especifica un formato

(tipo de datos) para cada uno de sus elementos de información (campos), junto con cualesquier restricciones que se apliquen a los elementos. (Las marcas cl.e. y * en la figura 10.9 se explicarán en la sección 10.4, cuando hablemos de la transformación ER-red.) Los tipos de datos que normalmente están disponibles dependen de los tipos que se pueden definir en el lenguaje de programación anfitrión. Supondremos que se pueden manejar cadenas de caracteres, números enteros y números con formato.†

Para especificar restricciones de clave sobre campos (o combinaciones de campos) que no pueden tener el mismo valor en más de un registro de un cierto tipo de registros, usamos la cláusula **DUPLICATES ARE NOT ALLOWED** (no se permiten duplicados). Por ejemplo, en la figura 10.10(a) NSS es una clave del tipo de registros EMPLEADO, y la combinación (NSSE, NÚMEROP) es una clave del tipo de registros TRABAJA_EN. Otras restricciones que podemos especificar para los campos se refieren a los valores que puede adoptar un campo numérico, mediante la cláusula **CHECK** (comprobar). Por ejemplo, podemos especificar que un campo numérico no puede tener un valor mayor que un cierto número.

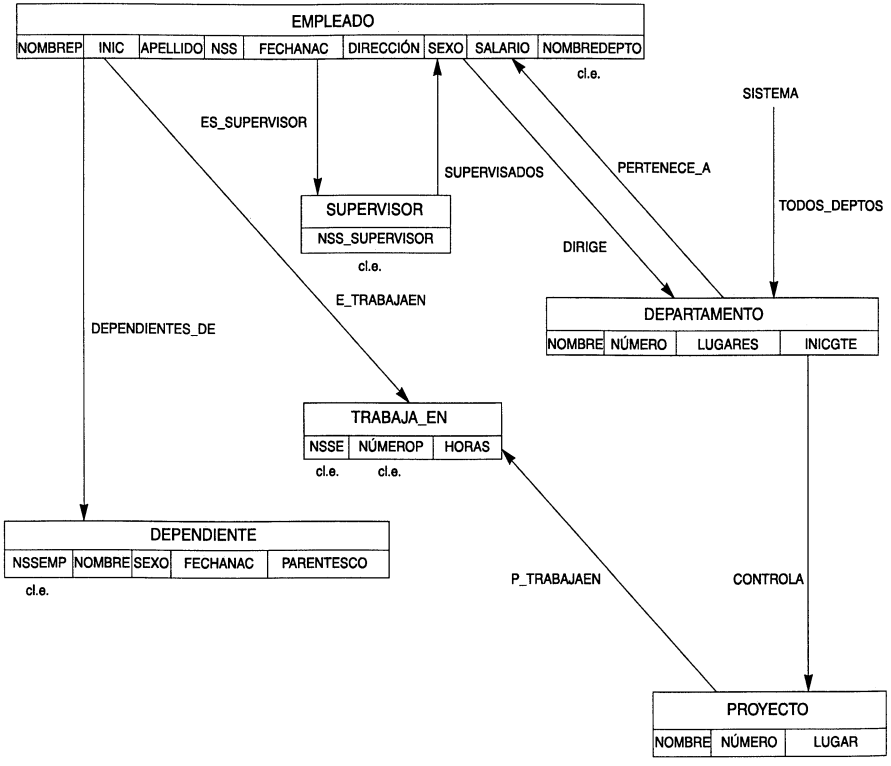


Figura 10.9 Esquema de red para la base de datos COMPAÑÍA.

†Los números con formato suelen especificarse con dos números (*i*,*j*), donde *i* es el número total de dígitos que tiene el número y *j* es el número de dígitos que siguen al punto decimal; tienen el mismo tamaño que una cadena de caracteres de tamaño *i*+1 (se requiere un carácter para el punto decimal). Un formato (7,2) corresponde a números de la forma *ddddd.dd*, donde cada *d* representa un dígito decimal.


```
SCHEMA NAME IS COMPAÑÍA

RECORD NAME IS EMPLEADO
    DUPLICATES ARE NOT ALLOWED FOR NSS
    DUPLICATES ARE NOT ALLOWED FOR NOMBREP, INIC, APELLIDO
        NOMBREP          TYPE IS    CHARACTER 15
        INIC             TYPE IS    CHARACTER 1
        APELLIDO         TYPE IS    CHARACTER 15
        NSS              TYPE IS    CHARACTER 9
        FECHANAC         TYPE IS    CHARACTER 9
        DIRECCIÓN        TYPE IS    CHARACTER 30
        SEXO             TYPE IS    CHARACTER 1
        SALARIO          TYPE IS    CHARACTER 10
        NOMBREDEPTO      TYPE IS    CHARACTER 15

RECORD NAME IS DEPARTAMENTO
    DUPLICATES ARE NOT ALLOWED FOR NOMBRE
    DUPLICATES ARE NOT ALLOWED FOR NÚMERO
        NOMBRE           TYPE IS    CHARACTER 15
        NÚMERO           TYPE IS    NUMERIC INTEGER
          LUGARES        TYPE IS    CHARACTER 15   VECTOR
        INICGTE          TYPE IS    CHARACTER 9

RECORD NAME IS PROYECTO
    DUPLICATES ARE NOT ALLOWED FOR NOMBRE
    DUPLICATES ARE NOT ALLOWED FOR NÚMERO
        NOMBRE           TYPE IS    CHARACTER 15
        NÚMERO           TYPE IS    NUMERIC INTEGER
        LUGAR            TYPE IS    CHARACTER 15

RECORD NAME IS TRABAJA_EN
    DUPLICATES ARE NOT ALLOWED FOR NSSE, NÚMEROP
        NSSE             TYPE IS    CHARACTER 9
        NÚMEROP          TYPE IS    NUMERIC INTEGER
        HORAS            TYPE IS    NUMERIC (4,1)

RECORD NAME IS SUPERVISOR
    DUPLICATES ARE NOT ALLOWED FOR NSS_SUPERVISOR
        NSS_SUPERVISOR   TYPE IS    CHARACTER 9

RECORD NAME IS DEPENDIENTE
    DUPLICATES ARE NOT ALLOWED FOR NSSEMP, NOMBRE
        NSSEMP           TYPE IS    CHARACTER 9
        NOMBRE           TYPE IS    CHARACTER 15
        SEXO             TYPE IS    CHARACTER 1
        FECHANAC         TYPE IS    CHARACTER 9
        PARENTESCO       TYPE IS    CHARACTER 10
```

Figura 10.10 (a) Declaraciones de tipos de registros para el esquema de la figura 10.9.
(*continúa en la página siguiente*)

```
SET NAME IS TODOS_DEPTOS
    OWNER IS SYSTEM
        ORDER IS SORTED BY DEFINED KEYS
    MEMBER IS DEPARTAMENTO
        KEY IS ASCENDING NOMBRE

SET NAME IS PERTENECE_A
    OWNER IS DEPARTAMENTO
        ORDER IS SORTED BY DEFINED KEYS
    MEMBER IS EMPLEADO
        INSERTION IS MANUAL
        RETENTION IS OPTIONAL
        KEY IS ASCENDING APELLIDO, NOMBREP, INIC
        CHECK IS NOMBREDEPTO IN EMPLEADO = NOMBRE IN DEPARTAMENTO

SET NAME IS CONTROLA
    OWNER IS DEPARTAMENTO
        ORDER IS SORTED BY DEFINED KEYS
    MEMBER IS PROYECTO
        INSERTION IS AUTOMATIC
        RETENTION IS MANDATORY
        KEY IS ASCENDING NOMBRE
        SET SELECTION IS BY APPLICATION

SET NAME IS DIRIGE
    OWNER IS EMPLEADO
        ORDER IS SYSTEM DEFAULT
    MEMBER IS DEPARTAMENTO
        INSERTION IS AUTOMATIC
        RETENTION IS MANDATORY
        SET SELECTION IS BY APPLICATION

SET NAME IS P_TRABAJAEN
    OWNER IS PROYECTO
        ORDER IS SYSTEM DEFAULT
        DUPLICATES ARE NOT ALLOWED
    MEMBER IS TRABAJA_EN
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        KEY IS NSSE
        SET SELECTION IS STRUCTURAL NÚMERO IN PROYECTO = NÚMEROP IN
        TRABAJA_EN

SET NAME IS E_TRABAJAEN
    OWNER IS EMPLEADO
        ORDER IS SYSTEM DEFAULT
        DUPLICATES ARE NOT ALLOWED
    MEMBER IS TRABAJA_EN
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        KEY IS NÚMEROP
        SET SELECTION IS STRUCTURAL NSS IN EMPLEADO = NSSE IN
        TRABAJA_EN
```

Figura 10.10 (b) Declaraciones de tipos de conjuntos para el esquema de la figura 10.9. (*continúa en la página siguiente*)


```
SET NAME IS SUPERVISADOS
    OWNER IS SUPERVISOR
        ORDER IS BY DEFINED KEY
        DUPLICATES ARE NOT ALLOWED
    MEMBER IS EMPLEADO
        INSERTION IS MANUAL
        RETENTION IS OPTIONAL
        KEY IS APELLIDO, INIC, NOMBREP

SET NAME IS ES_SUPERVISOR
    OWNER IS EMPLEADO
        ORDER IS SYSTEM DEFAULT
        DUPLICATES ARE NOT ALLOWED
    MEMBER IS SUPERVISOR
        INSERTION IS AUTOMATIC
        RETENTION IS MANDATORY
        KEY IS NSS_SUPERVISOR
        SET SELECTION IS BY VALUE OF NSS IN EMPLEADO
        CHECK IS NSS_SUPERVISOR IN SUPERVISOR = NSS IN EMPLEADO

SET NAME IS DEPENDIENTES_DE
    OWNER IS EMPLEADO
        ORDER IS BY DEFINED KEY
        DUPLICATES ARE NOT ALLOWED
    MEMBER IS DEPENDIENTE
        INSERTION IS AUTOMATIC
        RETENTION IS FIXED
        KEY IS ASCENDING NOMBRE
        SET SELECTION IS STRUCTURAL NSS IN EMPLEADO = NSSEMP
        IN DEPENDIENTE
```

Figura 10.10 (*continuación*) (b) Declaraciones de tipos de conjuntos para el esquema de la figura 10.9.

10.3.2 *Declaraciones de tipos de conjuntos y opciones de selección de conjuntos*

La figura 10.10(b) muestra declaraciones en DDL de red para los tipos de conjuntos del esquema COMPAÑÍA de la figura 10.9. Éstas son más complejas que las declaraciones de tipos de registros, porque hay más opciones disponibles. Cada tipo de conjuntos se nombra con la cláusula SET NAME IS (el nombre del conjunto es). Las opciones (restricciones) de inserción y retención que analizamos en la sección 10.2 se especifican para cada tipo de conjuntos mediante las cláusulas INSERTION IS (la inserción es) y RETENTION IS (la retención es). Si la opción de inserción es AUTOMATIC, también deberemos especificar la forma en que el sistema *seleccionará automáticamente una ocurrencia de conjunto* a la cual conectará un nuevo registro miembro cuando éste se inserte en la base de datos. La cláusula SET SELECTION (selección de conjunto) sirve para este fin. Tres métodos comunes de especificar la selección de conjunto son:

- SET SELECTION IS STRUCTURAL: Podemos especificar la selección de conjunto con los valores de dos campos que deben coincidir: un campo del tipo de registros propietario y uno del tipo de registros miembro. Esto se denomina restricción estructural en la terminología del modelo de red. Ejemplos son las declaraciones de los tipos de conjuntos P_TRABAJAEN y E_TRABAJAEN de la figura 10.10(b). El campo especificado del

tipo de registros propietario debe tener la restricción DUPLICATES ARE NOT ALLOWED (no se permiten duplicados) para que especifique un solo registro propietario y por tanto una sola ocurrencia de conjunto.

- SET SELECTION IS BY APPLICATION: La ocurrencia de conjunto la determina el programa de aplicación, que deberá hacer que la ocurrencia deseada sea el actual de conjunto (véase la Sec. 10.5.1) antes de almacenarse el nuevo registro miembro. Así, éste se conectará automáticamente a la ocurrencia de conjunto actual. Un ejemplo es el conjunto DIRIGE de la figura 10.10(b); para conectar un registro EMPLEADO a un DEPARTAMENTO como gerente de ese departamento, primero debemos hacer que ese registro EMPLEADO sea el actual de conjunto para el tipo de conjuntos DIRIGE. Si entonces almacenamos un nuevo registro DEPARTAMENTO, éste se conectará *automáticamente* a su registro EMPLEADO gerente como propietario.

- SET SELECTION IS BY VALUE OF <nombre de campo> IN <nombre de tipo de registros>: Una tercera opción es especificar un campo del tipo de registros propietario cuyo valor servirá para especificar una ocurrencia de conjunto al identificar el registro propietario del conjunto. Un ejemplo es el tipo de conjuntos ES_SUPERVISOR declarado en la figura 10.10(b), donde debemos asignar al campo NSS de la variable de programa UWA (véase la Sec. 10.5.1) correspondiente a EMPLEADO el valor del registro propietario deseado antes de almacenar un nuevo registro SUPERVISOR. El campo especificado en el tipo de registro propietario deberá tener la restricción DUPLICATES ARE NOT ALLOWED para que identifique un registro propietario único y por tanto una ocurrencia de conjunto única.

Otra opción para los conjuntos es especificar cómo se van a ordenar los registros miembro individuales en un ejemplar del conjunto, como vimos en la sección 10.2.4. Esto es importante dada la naturaleza de registro por registro del DML de red. La cláusula **ORDER IS** (el orden es) sirve para este fin, a veces aunada a la cláusula **KEY IS** (la clave es). Entre las opciones de la cláusula ORDER IS están las siguientes:

- ORDER IS SORTED BY DEFINED KEYS (el orden es según las claves definidas): Usamos la cláusula KEY IS para especificar uno o más campos del tipo de registros miembro; el sistema usa los valores de estos campos para ordenar los registros miembro dentro de cada ejemplar de conjunto. La cláusula KEY IS también especifica si los registros se deben ordenar de manera ascendente o descendente (ASCENDING o DESCENDING). Un ejemplo es el tipo de conjuntos PERTENECE_A, donde los registros EMPLEADO propiedad de un DEPARTAMENTO se ordenan según los valores ascendentes de APELLIDO, NOMBREP e INIC.

- ORDER IS FIRST (o LAST): Un nuevo registro miembro se inserta como el primer registro (o el último) de la ocurrencia de conjunto.

- ORDER IS BY SYSTEM DEFAULT (el orden se deja al sistema): No se especifica ningún orden en particular para los registros miembros de un ejemplar de conjunto.

- ORDER IS NEXT (o PRIOR): Un registro miembro nuevo se inserta inmediatamente después (o antes) del registro actual del conjunto. El programa debe hacer que el actual de conjunto (véase la Sec. 10.5.1) apunte al registro específico después (o antes) del cual se desea insertar el registro nuevo en el conjunto.

Otra cláusula que funciona en conjunción con la cláusula KEY IS es **DUPLICATES ARE NOT ALLOWED**. Ambas cláusulas se aplican a tipos de registros miembro dentro de los conjuntos. Esta combinación especifica que dos registros miembro *dentro de una ocurrencia de conjunto* no pueden tener los mismos valores en aquellos de sus campos que se declararon como claves. Un ejemplo es el campo NSSE que se declara como clave del tipo de conjuntos P_TRABAJAEN, lo que significa que dos registros TRABAJA_EN *dentro de la misma ocurrencia de conjunto* de P_TRABAJAEN no pueden tener el mismo valor de NSSE. Esto ha de especificarse porque no queremos conectar dos veces al mismo empleado como trabajador en el mismo proyecto.

Por último, consideremos la cláusula **CHECK**, que sirve para especificar una restricción estructural entre los registros propietario y miembro dentro de una ocurrencia de conjunto. Esto se usa con los conjuntos declarados MANUAL para especificar la condición de que ciertos campos de un registro miembro deben tener los mismos valores que ciertos campos del registro propietario. Si se intenta conectar un registro miembro que no satisfaga la condición de la cláusula CHECK, el sistema generará una condición de excepción y no conectará el registro miembro. Un ejemplo es la declaración del tipo de conjuntos PERTENECE_A en la figura 10.10(b). Esta restricción es similar a SET SELECTION IS STRUCTURAL, que se utiliza con conjuntos AUTOMATIC.

10.4 Uso de la transformación ER-red para el diseño de bases de datos de red*

Ahora veremos cómo un diseño conceptual de base de datos especificado como esquema ER (véase el Cap. 3) puede transformarse a un esquema de red. Usaremos el esquema ER COMPAÑÍA de la figura 3.2 para ilustrar nuestro análisis. En un esquema de red podemos representar explícitamente un tipo de vínculos como un tipo de conjuntos si es 1:*N*; sin embargo, no existe representación explícita si es 1:1 o M:N. Un método sencillo para representar un tipo de vínculos 1:1 consiste en usar un tipo de conjuntos pero hacer que cada ejemplar de conjunto tenga como máximo un registro miembro. Esta restricción la deben imponer los programas que actualizan la base de datos, ya que el SGBD no lo hace. En el caso de los tipos de vínculos M:N, la representación estándar es usar dos tipos de conjuntos y un tipo de registros de enlace. El modelo de red permite campos vectoriales y grupos repetitivos, con los que es posible representar directamente atributos compuestos y multivaluados, o incluso tipos de entidades débiles, como habremos de ver.

El esquema de red COMPAÑÍA de la figura 10.9 se puede derivar del esquema ER de la figura 3.2 mediante el siguiente procedimiento general de transformación. Ilustraremos cada paso con ejemplos del esquema COMPAÑÍA.

PASO 1: Entidades normales: Por cada tipo de entidades normal *E* del esquema ER, crear un tipo de registros *R* en el esquema de red. Todos los atributos simples (o compuestos) de *E* se incluyen como campos simples (o compuestos) de *R*. Todos los atributos multivaluados de *E* se incluyen como campos vectoriales o grupos repetitivos de *R*.

En nuestro ejemplo creamos los tipos de registros EMPLEADO, DEPARTAMENTO y PROYECTO e incluimos todos sus campos, como se muestra en la figura 10.9, con excepción de los que se marcan con cl.e. (clave externa) o con * (atributo de vínculo). Observe que el campo

LUGARES del tipo de registros DEPARTAMENTO es un campo vectorial porque representa un atributo multivaluado.

PASO 2: Entidades débiles: Para cada tipo de entidades débiles ED con el tipo de entidades identificador EI, o bien (a) creamos un tipo de registros *D* que represente a ED, haciendo a *D* el tipo de registros miembro de un tipo de conjuntos que relaciona *D* con el tipo de registros que representa a EI como propietario, o bien (b) creamos un grupo repetitivo en el tipo de registros que representa a EI para que represente los atributos de ED. Si escogemos la primera alternativa, el campo clave del tipo de registros que representa a EI se puede repetir en *D*.

En la figura 10.9 elegimos la primera alternativa; creamos un tipo de registros DEPENDIENTE y lo hacemos el tipo de registros miembro del tipo de conjuntos DEPENDIENTES_DE, propiedad de EMPLEADO. Duplicamos la clave NSS de EMPLEADO en DEPENDIENTE y la llamamos NSSEMP.

PASO 3: Vínculos uno-a-uno y uno-a-muchos: Para cada tipo de vínculos 1:1 o 1:*N* binario, no recursivo, *I* entre los tipos de entidades *E*1 y *E*2, creamos un tipo de conjuntos que relaciona los tipos de registros *R*1 y *R*2 que representan a *E*1 y *E*2, respectivamente. En el caso de un tipo de vínculos 1:1, elegimos arbitrariamente a *R*1 o a *R*2 como propietario y al otro como miembro; sin embargo, es preferible escoger como miembro un tipo de registros que represente una participación total en el tipo de vínculos. Otra opción para transformar un tipo de vínculos binario 1:1 *I* entre *E*1 y *E*2 consiste en crear un solo tipo de registros *R* que combine a *E*1, *E*2 e *I*, e incluya todos sus atributos; esto resulta útil si ambas participaciones, de *E*1 y *E*2, en *I* son totales y ni *E*1 ni *E*2 participan en muchos otros tipos de vínculos.

En el caso de un tipo de vínculos 1:*N*, se escoge como propietario el tipo de registros *R*1 que representa al tipo de entidades *E*1 en el lado 1 del tipo de vínculos, y como miembro se elige el tipo de registros *R*2 que representa al tipo de entidades *E*2 en el lado *N* del tipo de vínculos. Cualesquier atributos del tipo de vínculos *I* se incluyen como campos en el tipo de registros *miembro R*2.

En general, podemos duplicar arbitrariamente uno o más atributos de un tipo de registros propietario de un tipo de conjuntos —sea que represente un vínculo 1:1 o uno 1:*N*— en el tipo de registros miembro. Si el atributo duplicado es un atributo clave único del propietario, puede servir para declarar una restricción estructural sobre el tipo de conjuntos o para especificar la selección automática de propietario sobre la pertenencia al conjunto, como se explicó en la sección 10.2.1.

En nuestro ejemplo representamos el tipo de vínculos 1:1 DIRIGE de la figura 3.2 con el tipo de conjuntos DIRIGE, y escogemos DEPARTAMENTO como tipo de registros miembro debido a su participación total. El atributo FechaInic de DIRIGE se convierte en el campo INICGTE del tipo de registros miembro DEPARTAMENTO. Los dos tipos de vínculos 1:*N* no recursivos de la figura 3.2, PERTENECE_A y CONTROLA, se representan con los dos tipos de conjuntos PERTENECE_A y CONTROLA de la figura 10.9. En el caso del tipo de conjuntos PERTENECE_A, optamos por repetir un campo clave único, NOMBRE, del tipo de registros propietario DEPARTAMENTO en el tipo de registros miembro EMPLEADO, y lo llamamos NOMBRE-DEPTO. Decidimos no repetir ningún campo clave para el tipo de conjuntos CONTROLA. En general, un campo único de un tipo de registros propietario podría repetirse en el tipo de registros miembro.

PASO 4: Vínculos binarios de muchos-a-muchos: Por cada tipo de vínculos M:*N* binario *I* entre los tipos de entidades *E*1 y *E*2, creamos un tipo de registros de enlace X y hacemos que sea el tipo de registros miembro de dos tipos de conjuntos, cuyos propietarios son los tipos de registros *R*1 y *R*2 que representan a *E*1 y *E*2. Cualesquier atributos de *I* se convierten en campos de X. Si lo desea, el diseñador puede duplicar los campos únicos (clave) de los tipos de registros propietario como campos de X.

En la figura 10.9 creamos el tipo de registros de enlace TRABAJA_EN para representar el tipo de vínculos M:*N* TRABAJA_EN, e incluimos HORAS en él como campo. También creamos dos tipos de conjuntos E_TRABAJAEN y P_TRABAJAEN con TRABAJA_EN como tipo de registros miembro. Optamos por duplicar los campos clave únicos NSS y NÚMERO de los tipos de registros propietario EMPLEADO y PROYECTO en TRABAJA_EN, y los llamamos NSSE y NÚMEROP, respectivamente.

PASO 5: Vínculos recursivos: Por cada tipo de vínculos 1:1 o 1:*N* binario recursivo en el que el tipo de entidades *E* participe en ambos papeles, creamos un tipo de registros de enlace "ficticio" *V* y dos tipos de conjuntos que relacionen *V* con el tipo de registros X que representa a *E*. Se hará obligatorio que uno de los tipos de conjuntos, o ambos, tengan ejemplares de conjuntos con un solo registro miembro: uno en el caso de un tipo de vínculos 1:*N*, y los dos si se trata de un tipo de vínculos 1:1.

En la figura 3.2 tenemos un tipo 1:*N* recursivo, SUPERVISIÓN. Creamos el tipo de registros de enlace ficticio SUPERVISOR y los dos tipos de conjuntos ES_SUPERVISOR y SUPERVISADOS. Los programas de actualización de la base de datos obligan al tipo de conjuntos ES_SUPERVISOR a tener sólo un tipo de registros miembro en sus ejemplares de conjunto. Podemos considerar que cada registro miembro ficticio SUPERVISOR del tipo de conjuntos ES_SUPERVISOR representa a su registro EMPLEADO propietario *en el papel de supervisor*. El tipo de conjuntos SUPERVISADOS sirve para relacionar el registro SUPERVISOR "ficticio" con todos los registros EMPLEADO que representen a los empleados a los que supervisa directamente.

Los pasos anteriores consideran sólo tipos de vínculos binarios. El paso 6 muestra cómo se pueden transformar tipos de vínculos *n*-arios, con $n > 2$, mediante la creación de un tipo de registros de enlace, similar al caso del tipo de vínculos M:*N*.

PASO 6: Vínculos *n*-arios: Por cada tipo de vínculos *n*-ario *I*, con $n > 2$, creamos un tipo de registros de enlace X y hacemos que sea el tipo de registros miembro de *n* tipos de conjuntos. El propietario de cada tipo de conjuntos es el tipo de registros que representa a uno de los tipos de entidades que participan en el tipo de vínculos *I*. Cualesquier atributos de *I* se convertirán en campos de X. El diseñador puede, si lo desea, duplicar los campos únicos (clave) de los tipos de registros propietarios como campos de X.

Por ejemplo, consideremos el tipo de vínculos SUMINISTRA en el modelo ER que se muestra en la figura 10.11(a). Éste puede transformarse al tipo de registros SUMINISTRA y a los tres tipos de conjuntos que se muestran en la figura 10.11(b), donde decidimos no duplicar ningún campo de los propietarios.

Recuerde que es posible representar directamente los atributos compuestos y multivaluados en el modelo de red. Por añadidura, podemos representar los tipos de entidades débiles como tipos de registros individuales o bien como grupos repetitivos dentro del propietario; esto último resulta útil si el tipo de entidades débil no participa en otros tipos de vínculos. Si duplicamos un campo único (clave) del tipo de registros propietario en el tipo de registros miembro, podremos especificar una restricción estructural sobre la pertenencia a conjuntos

o la selección automática de conjunto; el SGBD conectará un registro miembro a un ejemplar de conjunto sólo si el mismo valor de campo clave está almacenado en los registros propietario y miembro. Esto equivale a hacer que el SGBD imponga la restricción automáticamente. Aunque no es obligatorio duplicar un campo clave coincidente del tipo de registros propietario en el tipo de registros miembro, es una práctica recomendable. El costo es el espacio de almacenamiento adicional que ocupa el campo repetido en cada registro miembro. Los beneficios son la imposición automática de la restricción y la disponibilidad del campo repetido en el registro miembro sin tener que obtener primero su propietario.

Al duplicar los campos clave de los registros propietarios en los registros miembros para todos los tipos de conjuntos de un esquema de red, ¡creamos tipos de registros que son prácticamente idénticos a las relaciones de un esquema de base de datos relacional! Las únicas diferencias se dan en el caso de los tipos de vínculos recursivos, los atributos multivaluados y los tipos de entidades débiles. En el caso de los tipos de vínculos 1:1 o 1:*N* recursivos, no hace falta crear una relación ficticia en el esquema relacional, como sucede en el modelo de red. En el caso de los tipos de entidades débiles y los atributos multivaluados, no necesitamos crear tipos de registros adicionales en el esquema de red, como sucede en el modelo relacional. ■

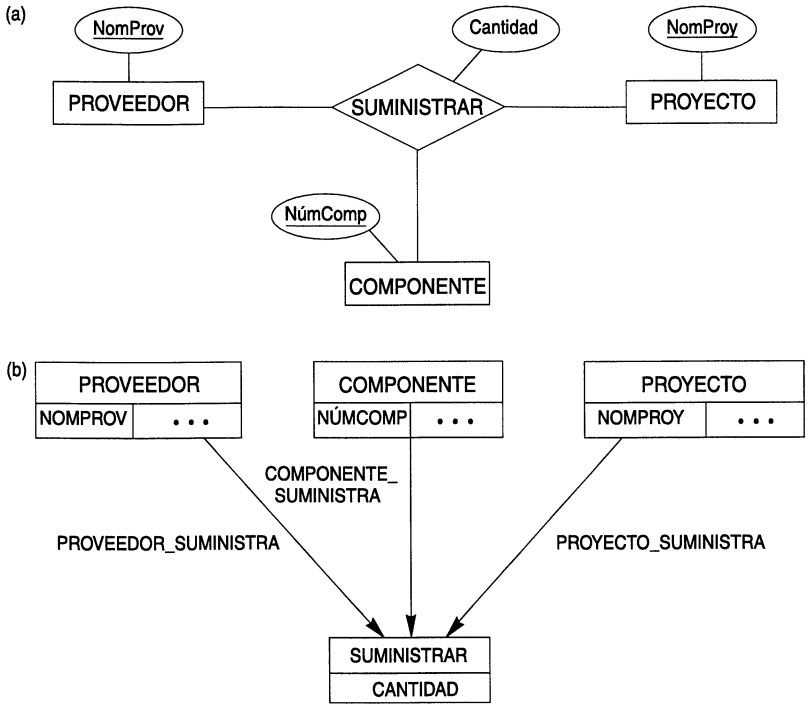


Figura 10.11 Transformación del tipo de vínculos *n*-ario ($n = 3$) SUMINISTRAR del modelo ER al modelo de red. (a) El modelo ER. (b) El modelo de red.

10.5 Programación de una base de datos de red*

En esta sección veremos cómo escribir programas que manipulen una base de datos de red, para realizar tareas como buscar y leer registros de la base de datos; insertar, eliminar y modificar registros, y conectar y desconectar registros de ocurrencias de conjuntos. Para ello, nos valdremos de un **lenguaje de manipulación de datos** (DML: *data manipulation language*). El DML asociado al modelo de red consiste en órdenes de registro por registro incorporadas en un lenguaje de programación de aplicación general denominado **lenguaje anfitrión.**† En la práctica, los lenguajes anfitrión de más amplia difusión son COBOL†† y PL/1, pero en nuestros ejemplos escribiremos segmentos de programas en la notación de PASCAL aumentada con las órdenes del DML de red.

10.5.1 *Conceptos básicos para la manipulación de bases de datos de red*

Antes de escribir programas para manipular una base de datos de red, necesitamos estudiar algunos conceptos básicos relacionados con la forma de escribir programas de manipulación de datos. El sistema de base de datos y el lenguaje de programación anfitrión son dos sistemas de software distintos que se enlazan mediante una interfaz común y se comunican exclusivamente a través de dicha interfaz. Como las órdenes del DML son de registro por registro, es necesario identificar registros específicos de la base de datos como **registros actuales.** El propio SGBD lleva el control de varios registros y ocurrencias de conjunto actuales por medio de un mecanismo denominado **indicadores de actualidad** (*currency indicators*). Además, el lenguaje de programación anfitrión requiere variables de programa locales para contener los registros de diferentes tipos de registros de modo que el programa anfitrión pueda manipularlos. El conjunto de estas variables locales en el programa suele denominarse **área de trabajo del usuario** (UWA: *user work area*). La comunicación entre el SGBD y el lenguaje de programación anfitrión se realiza principalmente a través de los indicadores de actualidad y del área de trabajo del usuario.

En esta sección estudiaremos estos dos conceptos. Nuestros ejemplos se referirán al esquema de base de datos de red que se muestra en la figura 10.9, que es la versión de red del esquema COMPAÑÍA utilizado en capítulos anteriores.

El área de trabajo del usuario (UWA). El UWA es un conjunto de variables de programa, declaradas en el programa anfitrión, que sirven para comunicar el contenido de registros individuales entre el SGBD y el programa anfitrión. Por cada tipo de registros del esquema en la base de datos, se debe declarar en el programa una variable correspondiente con el mismo formato. Se acostumbra usar los mismos nombres de tipos de registros y los mismos nombres de campos en las variables UWA y en el esquema de base de datos. De hecho, es posible declarar automáticamente las variables UWA en el programa empleando un paquete de software que crea variables de programa equivalentes a los tipos de registros declarados en el DDL de un esquema de base de datos.

En el caso del esquema COMPAÑÍA de la figura 10.9, si contáramos con una interfaz entre PASCAL y el SGBD de red, ésta podría crear las variables de programa PASCAL que aparecen en la figura 10.12. Es posible copiar de la base de datos o escribir en ella un registro

†Las órdenes de DML incorporadas también se denominan **sublenguaje de datos.**
††El DML CODASYL del informe DBTG fue propuesto originalmente como un sublenguaje de datos para COBOL.

individual de cada tipo de registros haciendo uso de la variable de programa correspondiente del UWA. La orden GET (obtener; véase la Sec. 10.5.3) lee físicamente un registro y lo copia en la variable de programa correspondiente; así, podemos hacer referencia a los valores de los campos para imprimirlos o usarlos en cálculos. Si queremos escribir un registro en la base de datos, primero asignamos los valores de sus campos a los campos de la variable de programa y luego, con la orden STORE (almacenar; véase la Sec. 10.5.3), guardamos físicamente ese registro en la base de datos.

Indicadores de actualidad. En el DML de red, para las obtenciones y actualizaciones se realizan desplazándose por los registros de la base de datos, acción a la que se llama **navegación** (o **recorrido**) por los registros; por ello, resulta crucial seguir la pista de las búsquedas. Los indicadores de actualidad son un medio con el cual el SGBD puede mantenerse al tanto de los registros y ocurrencias de conjunto a los que se tuvo acceso más recientemente. Desempeñan el papel de marcadores de posición para poder procesar registros nuevos a partir de los últimos que se usaron, hasta obtener todos los registros que contienen la información buscada. Podemos concebir a cada indicador de actualidad como un apuntador a registro (o una dirección de registro) que apunta a un solo registro de la base de datos. En un SGBD de red se utilizan varios indicadores de actualidad:

- **Actual de tipo de registros:** Por cada tipo de registros, el SGBD sigue la pista al último registro de ese tipo al que se tuvo acceso. Si todavía no se ha leído ningún registro de ese tipo, el registro actual no está definido.
- **Actual de tipo de conjuntos:** Por cada tipo de conjuntos en el esquema, el SGBD sigue la pista a la última ocurrencia de conjunto de ese tipo de conjuntos a la que se tuvo acceso. La ocurrencia se especifica con un solo registro de ese conjunto, que es *el propietario o uno de los registros miembro*. Por tanto, el actual de conjunto (o conjunto actual) apunta a un registro, aunque se use para seguir la pista de una ocurrencia de conjunto. Si el programa no ha tenido acceso a ningún registro de ese tipo de conjuntos, el actual de conjunto no está definido.
- **Actual de unidad de ejecución (CRU: *current of run unit*):** Una unidad de ejecución es un programa de acceso a la base de datos que el computador está ejecutando. Por cada unidad de ejecución, el CRU sigue la pista al último registro al que tuvo acceso el programa; este registro puede ser de *cualquier* tipo de registros en la base de datos.

Cada vez que un programa ejecuta una orden de DML, el SGBD actualiza los indicadores de actualidad para los tipos de registros y tipos de conjuntos afectados por esa orden. Es preciso entender perfectamente la forma en que cada orden de DML afecta los indicadores de actualidad. Muchas órdenes de DML afectan dichos indicadores y también dependen de ellos. En la sección 10.5.3 ilustraremos la forma en que las diferentes órdenes de DML afectan los indicadores de actualidad.

Indicadores de estado. Hay varios **indicadores de estado** que devuelven una indicación de éxito o fracaso después de ejecutarse cada orden de DML. El programa puede revisar los valores de estos indicadores de estado y emprender las acciones apropiadas: ya sea continuar la ejecución o transferir el control a una rutina de manejo de errores.

Llamaremos DB_STATUS a la principal variable de estado y supondremos que se declara implícitamente en el programa anfitrión. Después de cada orden de DML, el valor de DB_STATUS indicará si la orden tuvo éxito o si hubo un error o una excepción. La excepción


```
type REGISTROLUGAR   = (* esto es para el campo vector LUGARES de DEPARTAMENTO *)
                       record
                       LUGAR : packed array [1..15] of char;
                       NEXT : ^REGISTROLUGAR
                       end;
var EMPLEADO         : record
                       NOMBREP : packed array [1..15] of char;
                       INIC : char;
                       APELLIDO : packed array [1..15] of char;
                       NSS : packed array [1..9] of char;
                       FECHANAC : packed array [1..9] of char;
                       DIRECCIÓN : packed array [1..30] of char;
                       SEXO : char;
                       SALARIO : packed array [1..10] of char;
                       NOMBREDEPTO : packed array [1..15] of char;
                       end;
DEPARTAMENTO         : record
                       NOMBRE : packed array [1..15] of char;
                       NÚMERO : integer;
                       LUGARES : ^REGISTROLUGAR;
                       INICGTE : packed array [1..9] of char;
                       end;
PROYECTO             : record
                       NOMBRE : packed array [1..15] of char;
                       NÚMERO : integer;
                       LUGAR : packed array [1..15] of char;
                       end;
TRABAJA_EN           : record
                       NSSE : packed array [1..9] of char;
                       NÚMEROP : integer;
                       HORAS : packed array [1..4] of char;
                       end;
SUPERVISOR           : record
                       NSS_SUPERVISOR : packed array [1..9] of char;
                       end;
DEPENDIENTE          : record
                       NSSEMP : packed array [1..9] of char;
                       NOMBRE : packed array [1..15] of char;
                       SEXO : char;
                       FECHANAC : packed array [1..9] of char;
                       PARENTESCO: packed array [1..10] of char
                       end;
```

Figura 10.12 Variables de programa en PASCAL para la UWA correspondiente al esquema de red de la figura 10.9.

más común es la de FIN_DE_CONJUNTO (**EOS:** *end of set*). Esto no es un error; sólo indica que no hay más registros miembro en una ocurrencia de conjunto. Por ello, a menudo se le utiliza para terminar un ciclo de programa que procesa todos los elementos miembro de un ejemplar de conjunto. Una orden de DML para buscar el siguiente miembro de un conjunto (o el anterior) devuelve una excepción EOS cuando no existe dicho miembro. El programa

verifica que DB_STATUS = EOS para terminar el ciclo. Supondremos que un valor de cero en DB_STATUS indica que la orden se ejecutó con éxito sin que hubiera excepciones.

Ilustración de los indicadores de actualidad y de la UWA. Supongamos que un programa ejecuta órdenes de base de datos que originan los siguientes sucesos en los ejemplares de base de datos que se muestran en la figura 10.7(f):

- Se tiene acceso al registro EMPLEADO E3.
- Siguiendo el apuntador PRIMERO(E_T) de E3, se tiene acceso al registro TRABAJA_EN T4; continuando con los apuntadores SIGUIENTE(E_T) de los registros TRABAJA_EN, se tiene acceso a T5 y T6.
- Se obtiene el registro T6 colocándolo en la variable UWA correspondiente.

La figura 10.13 ilustra los efectos de estos acontecimientos sobre las variables UWA y los indicadores de actualidad del SGBD cuando se aplican a los ejemplares de la figura 10.7(f).

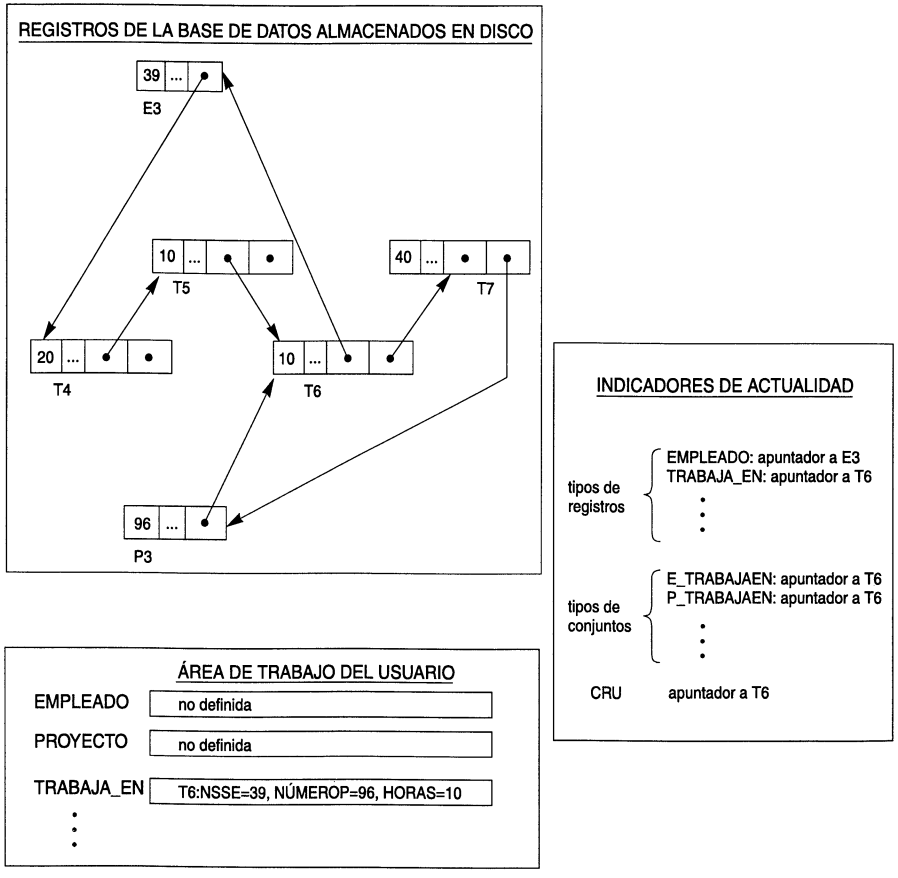


Figura 10.13 Variables UWA e indicadores de actualidad.

Paso	Actuales de registros					Actuales de conjuntos					CRU
	EMPLEADO	TRABAJA_EN	PROYECTO	DEPENDIENTE	...	E_TRABAJAEN (E_T)	P_TRABAJAEN (P_T)	DEPENDIENTES_DE	CONTROLA	...	
BUSCAR E3	^E3					^E3		^E3			^E3
BUSCAR T6 (miembro de E_T)	^E3	^T6				^T6	^T6	^E3			^T6
BUSCAR P3 (propietario de P_T)	^E3	^T6	^P3			^T6	^P3	^E3	^P3		^P3

Figura 10.14 Ejemplo de cómo cambian los indicadores de actualidad.

La figura 10.14 muestra cómo cambian los indicadores de actualidad conforme tienen lugar los acontecimientos, con un suceso adicional que localiza el registro propietario P3 de T6 en el conjunto P_T. En la figura 10.14 un apuntador a un registro *x* se denota con ^*x*. Observe que, después de la primera orden, el actual de conjunto de *todos* los tipos de conjuntos en los que EMPLEADO participa en la figura 10.9 apunta a E3; éstos son E_TRABAJAEN y DEPENDIENTES_DE, que aparecen en la figura 10.14, así como SUPERVISADOS, ES_SUPERVISOR, DIRIGE y PERTENECE_A, que no se muestran. El actual del tipo de registros EMPLEADO se mantiene durante las siguientes órdenes, pero el de DEPENDIENTE nunca se establece (sigue indefinido). Observe cómo cambia el actual para el tipo de conjuntos E_T conforme pasamos del propietario al miembro y para P_T conforme pasamos del miembro al propietario.

10.5.2 *Lenguaje de manipulación de datos (DML) de red*

Las órdenes para manipular una base de datos de red constituyen el DML de red. Estas órdenes suelen estar incorporadas en un lenguaje de programación de propósito general, al que llamamos anfitrión. Las órdenes de DML se pueden agrupar en órdenes de navegación, de obtención y de actualización. Las órdenes de **navegación** sirven para establecer los indicadores de actualidad a registros y ocurrencias de conjunto específicos de la base de datos. Las órdenes de **obtención** leen el registro actual de la unidad de ejecución (CRU). Las órdenes de **actualización** se pueden dividir en dos subgrupos: uno para actualizar registros y el otro para actualizar ocurrencias de conjuntos. Con las órdenes de actualización de registros se almacenan registros nuevos, se eliminan registros obsoletos y se modifican los valores de los campos, en tanto que con las órdenes de actualización de conjuntos se conecta o desconecta un registro miembro en una ocurrencia de conjunto o se pasa un registro miembro de una ocurrencia a otra. El conjunto completo de órdenes se resume en la tabla 10.2.

A continuación analizaremos todas estas órdenes de DML e ilustraremos nuestro análisis con ejemplos que aplican el esquema de red mostrado en la figura 10.9 y que se define con las declaraciones DDL de las figuras 10.10(a) y (b). Por lo general, las órdenes de DML que presentamos se basan en la propuesta CODASYL DBTG. PASCAL será el lenguaje anfitrión en nuestros ejemplos, pero los estudiantes pueden practicar escribiendo estos programas con otros lenguajes anfitrión. Los ejemplos consisten en segmentos de programas cortos sin declaraciones de variables. Supondremos que en otro lugar del programa en PASCAL ya se han definido las variables de UWA (área de trabajo del usuario) que se muestran en la figura 10.12. En nuestros programas, antepondremos a las órdenes de DML un signo $ para distinguirlas de las instrucciones de PASCAL. Escribiremos las palabras reservadas de PASCAL —como *if*, *then*, *while* y *for*— en minúsculas.

En nuestros ejemplos muchas veces tendremos que asignar valores a los campos de las variables UWA de PASCAL; emplearemos la notación de PASCAL para asignarlos. Por ejemplo, para asignar 'José' y 'Silva' a los campos NOMBREP y APELLIDO de la variable UWA EMPLEADO, escribiremos:

```
EMPLEADO.NOMBREP := 'José'; EMPLEADO.APELLIDO := 'Silva';
```

Adviértase que en el lenguaje de programación COBOL (para el cual se diseñó originalmente el DML CODASYL) las mismas asignaciones se escriben así:

```
MOVE 'José' TO NOMBREP IN EMPLEADO
MOVE 'Silva' TO APELLIDO IN EMPLEADO
```

10.5.3 *Órdenes de DML para obtención y navegación*

La orden de DML para obtener un registro es **GET**. Antes de emitir esta orden, el programa debe especificar como CRU el registro que desea leer, empleando para ello las órdenes de navegación **FIND** (buscar) apropiadas. Hay muchas variaciones de FIND; primero veremos cómo se usa esta orden para localizar ejemplares de registros de un tipo de registros y después veremos las variaciones para procesar ocurrencias de conjuntos.

Órdenes de DML para localizar registros de un tipo de registros. Hay dos variantes principales de la orden FIND para localizar un registro de un cierto tipo y hacerlo el CRU y el actual del tipo de registros. Otros indicadores de actualidad pueden resultar afectados, como veremos en breve. El formato de estas dos órdenes es como sigue, donde las partes opcionales de la orden aparecen entre corchetes, [...]:

- **FIND ANY** <nombre de tipo de registros> [**USING** <lista de campos>]
- **FIND DUPLICATE** <nombre de tipo de registros> [**USING** <lista de campos>]

Ahora ilustraremos con ejemplos el empleo de estas órdenes. Si deseamos leer el registro EMPLEADO correspondiente al empleado llamado José Silva e imprimir su salario, podemos escribir EJ1:

```
EJ1:  1   EMPLEADO.NOMBREP := 'José'; EMPLEADO.APELLIDO := 'Silva';
      2   $FIND ANY EMPLEADO USING NOMBREP, APELLIDO;
      3   if DB_STATUS = 0
      4       then begin
      5           $GET EMPLEADO;
      6           writeln (EMPLEADO.SALARIO)
      7           end
      8       else writeln ('no se halló el registro');
```

La orden **FIND ANY** (buscar cualquiera) busca en la base de datos el *primer* registro del <nombre de tipo de registros> especificado, tal que los valores de campos del registro coincidan con los valores antes asignados a los campos UWA correspondientes especificados en la cláusula USING de la orden.

En EJ1, las líneas 1 y 2 equivalen a decir: "buscar el primer registro EMPLEADO que satisfaga la condición NOMBREP = 'José' y APELLIDO = 'Silva' y convertirlo en el registro actual de la unidad de ejecución (CRU)". La orden GET equivale a decir: "obtener el registro CRU y colocarlo en la variable de programa UWA correspondiente". En general, siempre que se usa una orden FIND, el programa deberá comprobar si logró encontrar un registro evaluando DB_STATUS. Un valor de cero significa que se logró encontrar un registro, así que escribimos la instrucción if...then a partir de la línea 3 antes de emitir la orden GET en la línea 5 de EJ1.

Tabla 10.2 Resumen de las órdenes DML de red

(OBTENCIÓN)	
GET	OBTENER EL ACTUAL DE UNIDAD DE EJECUCIÓN (CRU) Y COLOCARLO EN LA VARIABLE CORRESPONDIENTE DEL ÁREA DE TRABAJO DEL USUARIO
(NAVEGACIÓN)	
FIND	BUSCAR. RESTABLECE LOS INDICADORES DE ACTUALIDAD; SIEMPRE ESTABLECE EL CRU; TAMBIÉN ESTABLECE LOS INDICADORES DE ACTUALIDAD DE LOS TIPOS DE REGISTROS Y TIPOS DE CONJUNTOS QUE INTERVIENEN. HAY MUCHAS VARIACIONES DE FIND.
(ACTUALIZACIÓN DE REGISTROS)	
STORE	ALMACENAR EL REGISTRO NUEVO EN LA BASE DE DATOS Y CONVERTIRLO EN EL CRU.
ERASE	ELIMINAR DE LA BASE DE DATOS EL REGISTRO QUE ES EL CRU.
MODIFY	MODIFICAR ALGUNOS CAMPOS DEL REGISTRO QUE ES EL CRU.
(ACTUALIZACIÓN DE CONJUNTOS)	
CONNECT	CONECTAR UN REGISTRO MIEMBRO (EL CRU) A UN EJEMPLAR DE CONJUNTO.
DISCONNECT	ELIMINAR UN REGISTRO MIEMBRO (EL CRU) DE UN EJEMPLAR DE CONJUNTO.
RECONNECT	PASAR UN REGISTRO MIEMBRO (EL CRU) DE UN EJEMPLAR DE CONJUNTO A OTRO.

La orden FIND no sólo establece el CRU, sino también otros indicadores de actualidad, a saber, los del tipo de registros cuyo nombre se especifica en la orden y los de cualesquier tipos de conjuntos en los que ese tipo de registros participe como propietario o miembro. Por ello, la orden FIND anterior también establece los indicadores de actualidad del tipo de registros EMPLEADO y de todos los tipos de conjuntos en los que el registro localizado interviene como propietario o miembro de una ocurrencia de conjunto. No obstante, la orden GET siempre obtiene el CRU, *el cual puede no ser igual al actual del tipo de registros.*[†] El sistema IDMS combina las instrucciones FIND y GET en una sola, llamada OBTAIN.

Merece la pena considerar dos variaciones de EJ1. Primera, si sustituimos la línea 5 por sólo $GET, obtendremos exactamente el mismo resultado que antes. La diferencia es que al incluir el nombre del tipo de registros en la orden GET —como en EJ1— el sistema comprobará que el CRU sea del tipo de registros especificado; si no es así, se generará un error y no

[†]Se ha sugerido una variación del DML de red que usa la orden GET para obtener *el registro actual del tipo de registros especificado*. Esto facilita la escritura de algunos programas, pero la mayoría de los SGBD de red emplean la orden GET para obtener el CRU, como se explica aquí.

se colocará el CRU en la variable UWA. Como segunda variación, si sustituimos la línea 5 por, digamos, $GET DEPARTAMENTO, se generará un error porque el tipo de registros especificado en la orden GET, DEPARTAMENTO, no coincidirá con el tipo de registros del CRU, EMPLEADO.

Si dos o más registros satisfacen nuestra búsqueda y queremos obtenerlos todos, deberemos escribir una construcción cíclica en el lenguaje de programación anfitrión. Por ejemplo, para obtener todos los registros EMPLEADO de los empleados que trabajan en el departamento de investigación e imprimir todos sus nombres, podemos escribir EJ2.

```
EJ2:  EMPLEADO.NOMBREDEPTO := 'Investigación';
      $FIND ANY EMPLEADO USING NOMBREDEPTO;
      while DB_STATUS = 0 do
          begin
          $GET EMPLEADO;
          writeln ( EMPLEADO.NOMBREP, ' ', EMPLEADO.APELLIDO );
          $FIND DUPLICATE EMPLEADO USING NOMBREDEPTO
          end;
```

La orden **FIND DUPLICATE** (buscar duplicado) busca el *siguiente* registro, a partir del registro actual, que satisface la búsqueda. No podemos usar FIND ANY, porque éste siempre busca el primer registro que satisface la búsqueda. Cabe señalar que los registros "primero" y "siguiente" no tienen un significado especial aquí, ya que no especificamos nigún orden para los registros EMPLEADO en el DDL de la figura 10.10(b). El sistema busca los registros EMPLEADO físicamente en el orden en que están almacenados; sin embargo, una vez que se han revisado todos los registros en el ciclo *while*, el sistema asignará a DB_STATUS el valor correspondiente a la condición de excepción "no se encuentran más registros" y el ciclo concluirá.

Órdenes de DML para procesar conjuntos. Tenemos las siguientes variantes de FIND para procesar conjuntos:

- FIND (FIRST | NEXT | PRIOR | LAST | ...) <nombre de tipo de registros>
  WITHIN <nombre de tipo de conjuntos> [USING <nombres de campos>]
- FIND OWNER WITHIN <nombre de tipo de conjuntos>

Una vez que hayamos establecido una ocurrencia de conjunto actual de un tipo de conjuntos, podremos usar la orden FIND para localizar diversos registros que participen en la ocurrencia de conjunto. Es posible localizar el registro propietario o bien uno de los registros miembro y hacer que ese registro sea el CRU. Con **FIND OWNER** podemos localizar el registro propietario, y con **FIND FIRST, FIND NEXT, FIND LAST** o **FIND PRIOR** podemos localizar el primer registro miembro, el siguiente, el último o el anterior, respectivamente, del ejemplar de conjunto.

Recordemos que el indicador actual de conjunto puede estar apuntando al registro propietario o bien a cualquier miembro de una ocurrencia de conjunto. Las órdenes FIND OWNER, FIND FIRST y FIND LAST tienen el mismo efecto, independientemente del registro determinado en la ocurrencia de conjunto al que apunte el actual de conjunto. Sin embargo, FIND NEXT y FIND PRIOR *sí dependen* del actual de conjunto. En el caso de FIND NEXT, si el actual de conjunto es el propietario, se busca el primer miembro; si el actual de conjunto es cualquier registro miembro, excepto el último, se busca el siguiente registro miembro; por último, si el actual de conjunto es el último registro miembro del conjunto, se asigna a DB_STATUS el valor correspondiente a la excepción EOS (fin de conjunto). En el caso de FIND PRIOR (buscar anterior), se realizan acciones similares.

El siguiente ejemplo ilustra el empleo de FIND FIRST y FIND NEXT. La consulta consiste en imprimir, en orden alfabético por apellido, los nombres de los empleados que trabajan en el departamento de investigación; esto lo hace EJ3, que es similar a EJ2, excepto por el requisito de ordenamiento. EJ3 obtiene primero el registro DEPARTAMENTO de 'Investigación' y luego los registros EMPLEADO propiedad de ese registro a través del conjunto PERTENECE_A. Recuerde que, en la declaración del tipo de conjuntos PERTENECE_A de la figura 10.10(b), especificamos que los registros miembro de cada ejemplar de conjunto PERTENECE_A se almacenan en orden ascendente según los valores de APELLIDO, NOMBREP e INIC. Al obtener los registros miembro EMPLEADO en orden, podremos imprimir en orden alfabético los nombres de los empleados en EJ3. Observe cómo terminamos el ciclo verificando DB_STATUS. Una vez localizado el último registro miembro de la ocurrencia de conjunto, la siguiente orden FIND NEXT asigna a DB_STATUS el valor correspondiente a la excepción EOS (fin de conjunto).

```
EJ3:  DEPARTAMENTO.NOMBRE := 'Investigación';
      $FIND ANY DEPARTAMENTO USING NOMBRE;
      if DB_STATUS = 0 then
          begin
          $FIND FIRST EMPLEADO WITHIN PERTENECE_A;
          while DB_STATUS = 0 do
              begin
              $GET EMPLEADO;
              writeln ( EMPLEADO.APELLIDO, ' ', EMPLEADO.NOMBREP );
              $FIND NEXT EMPLEADO WITHIN PERTENECE_A
              end
          end;
```

El siguiente ejemplo ilustra el uso de FIND OWNER. La consulta consiste en imprimir el nombre y el número del proyecto y las horas por semana de todos los proyectos en los que trabaja el empleado José Silva (suponiendo que sólo hay un empleado que se llama así). Esto se muestra en EJ4. La orden FIND ANY establece el CRU así como el registro actual del tipo de registros EMPLEADO y el actual de conjunto del tipo de conjuntos E_TRABAJAEN. Después recorremos cíclicamente todos los registros miembro TRABAJA_EN del conjunto E_TRABAJEN actual, y dentro de cada ciclo buscamos el registro PROYECTO propietario del registro TRABAJA_EN a través del tipo de conjuntos P_TRABAJAEN, mediante la orden FIND OWNER (buscar propietario). Adviértase que no es preciso verificar DB_STATUS después de la orden FIND OWNER, porque la opción de retención para el conjunto P_TRABAJAEN es FIXED, lo que significa que todos los registros TRABAJA_EN deben pertenecer a un ejemplar de conjunto P_TRABAJAEN:

```
EJ4:  EMPLEADO.NOMBREP := 'José'; EMPLEADO.APELLIDO := 'Silva';
      $FIND ANY EMPLEADO USING NOMBREP, APELLIDO;
      if DB_STATUS = 0 then
          begin
          $FIND FIRST TRABAJA_EN WITHIN E_TRABAJAEN;
          while DB_STATUS = 0 do
              begin
              $GET TRABAJA_EN;
              $FIND OWNER WITHIN P_TRABAJAEN;
              $GET PROYECTO;
              writeln ( PROYECTO.NOMBRE, PROYECTO NÚMERO,
                        TRABAJA_EN.HORAS );
```

```
            $FIND NEXT TRABAJA_EN WITHIN E_TRABAJAEN
            end
        end;
```

En EJ3 y EJ4, procesamos todos los registros miembro de un ejemplar de conjunto. Como alternativa, podemos procesar selectivamente sólo los registros miembro que satisfagan alguna condición. Si la condición es una comparación de igualdad de uno o más campos, podemos anexar una cláusula USING (empleando) a la orden FIND. Como ilustración, consideremos la solicitud de imprimir los nombres de todos los empleados que trabajan horario completo —40 horas a la semana— en el proyecto 'ProductoX'; este ejemplo se ilustra como EJ5:

```
EJ5:  PROYECTO.NOMBRE := 'ProductoX';
      $FIND ANY PROYECTO USING NOMBRE;
      if DB_STATUS = 0 then
          begin
          TRABAJA_EN.HORAS := '40.0';
          $FIND FIRST TRABAJA_EN WITHIN P_TRABAJAEN USING HORAS;
          while DB_STATUS = 0 do
              begin
              $GET TRABAJA_EN;
              $FIND OWNER WITHIN E_TRABAJAEN; $GET EMPLEADO;
              writeln (EMPLEADO.NOMBREP, EMPLEADO.APELLIDO);
              $FIND NEXT TRABAJA_EN WITHIN P_TRABAJAEN USING HORAS
              end
          end;
```

En EJ5, la estipulación USING HORAS en FIND FIRST y FIND NEXT especifica que sólo se buscarán los registros TRABAJA_EN en el ejemplar de conjunto actual de P_TRABAJAEN cuyo valor de campo HORAS coincida con el valor en TRABAJA_EN.HORAS del UWA, al que se asignó el valor '40.0' en el programa. Observe que la cláusula USING con FIND NEXT sirve para buscar el *siguiente registro miembro dentro de la misma ocurrencia de conjunto*; cuando procesamos registros de un tipo de registros *independientemente de los conjuntos a los que pertenezcan*, empleamos FIND DUPLICATE en vez de FIND NEXT.

Si la condición que selecciona registros miembro específicos de un ejemplar de conjunto implica operadores de comparación *distintos de la igualdad*, como menor que o mayor que, tendremos que obtener cada registro miembro y verificar si satisface la condición en el programa anfitrión mismo. Recomendamos al lector modificar EJ4 de modo que sólo se obtengan los proyectos para los cuales el valor de TRABAJA_EN.HORAS sea mayor que 5. Esta condición deberá colocarse inmediatamente después de la obtención física del registro TRABAJA_EN.

Para procesar muchos conjuntos empleamos varios ciclos incorporados en el mismo segmento de programa. Por ejemplo, consideremos la siguiente consulta: Para cada departamento, imprimir el nombre del departamento y el de su gerente; y para cada empleado que pertenece a ese departamento, imprimir el nombre del empleado y la lista de nombres de los proyectos en los que trabaja.

Esta consulta nos obliga a procesar el conjunto TODOS_DEPTOS, propiedad del sistema, a fin de obtener los registros DEPARTAMENTO. Mediante el conjunto PERTENECE_A, el programa obtiene los registros EMPLEADO de cada DEPARTAMENTO. Luego, por cada empleado que se encuentre, se tendrá acceso al conjunto E_TRABAJAEN para localizar los registros TRABAJA_EN.

Para cada uno de estos registros encontrado, con una orden "FIND OWNER WITHIN P_TRABAJAEN" se localizará el PROYECTO apropiado.

Cómo emplear los recursos del lenguaje de programación anfitrión. Dado que el DML de red es un lenguaje de registro por registro, necesitaremos usar los recursos del lenguaje de programación anfitrión cada vez que una consulta requiera un conjunto de registros. También tendremos que usar el lenguaje anfitrión para calcular funciones sobre conjuntos de registros, como cuentas o promedios, mismas que el programador deberá implementar explícitamente. Esto contrasta con la facilidad para especificar tales funciones en lenguajes de alto nivel como SQL (Cap. 7) y QUEL (Cap. 8).

Un ejemplo final ilustra la forma de calcular funciones como CUENTA y PROMEDIO. Suponga que nos interesa calcular el número de empleados que son supervisores en cada departamento y su salario medio; esto se muestra en EJ6. Suponemos que en otro lugar del programa se declaró una función de PASCAL llamada convertir_en_real, que convierte el valor de cadena del campo SALARIO en un número real. También necesitamos declarar en otro lugar las variables de programa sal_total:real y núm_de_supervisores:integer para hacer la suma acumulada del salario total y del número de supervisores en cada departamento, respectivamente. En EJ6, observe cómo determinamos si un empleado es supervisor verificando si un registro EMPLEADO participa como propietario en algún ejemplar del conjunto ES_SUPERVISOR:

```
EJ6:  $FIND FIRST DEPARTAMENTO WITHIN TODOS_DEPTOS;
      while DB_STATUS = 0 do
          begin
          $GET DEPARTAMENTO;
          write (DEPARTAMENTO.NOMBRE);  (* nombre del departamento *)
          sal_total := 0; núm_de_supervisores := 0;
          $FIND FIRST EMPLEADO WITHIN PERTENECE_A;
          while DB_STATUS = 0 do
              begin
              $GET EMPLEADO;
              $FIND FIRST SUPERVISOR WITHIN ES_SUPERVISOR;
              (* el empleado es supervisor si posee un registro SUPERVISOR
              a través de ES_SUPERVISOR *)
              if DB_STATUS = 0 then (* probar si es supervisor *)
                  begin
                  sal_total := sal_total + convertir_en_real
                  (EMPLEADO.SALARIO);
                  núm_de_supervisores := núm_de_supervisores + 1
                    end;
              $FIND NEXT EMPLEADO WITHIN PERTENECE_A;
              end;
          writeln('número de supervisores =', núm_de_supervisores);
          writeln('salario medio de los supervisores =', sal_total/ núm_de
          _supervisores);
          writeln();
          $FIND NEXT DEPARTAMENTO WITHIN TODOS_DEPTOS
          end;
```

10.5.4 *Órdenes de DML para actualizar la base de datos*

Las órdenes de DML para actualizar una base de datos de red se resumen en la tabla 10.2. Aquí estudiaremos primero las órdenes con que se actualizan los registros, a saber, las órdenes STORE, ERASE y MODIFY, que sirven para insertar un registro nuevo, eliminar un registro y modificar algunos campos de un registro, respectivamente. A continuación, ilustraremos las órdenes con que podemos modificar ejemplares de conjuntos: CONNECT, DISCONNECT y RECONNECT.

La orden STORE. La orden **STORE** (almacenar) sirve para insertar un registro nuevo. Antes de emitir esta orden debemos preparar la variable UWA del tipo de registros correspondiente para que sus campos contengan los valores del registro nuevo. Por ejemplo, si deseamos insertar un nuevo registro EMPLEADO para Juan F. Silva, podemos usar EJ7:

```
EJ7:  EMPLEADO.NOMBREP := 'Juan';
      EMPLEADO.APELLIDO := 'Silva';
      EMPLEADO.INIC := 'F';
      EMPLEADO.NSS := '567342739';
      EMPLEADO.DIRECCIÓN := 'Ave. Nogal 40, Yautepec, Morelos 55433';
      EMPLEADO.FECHANAC := '10-ENE-55';
      EMPLEADO.SEXO := 'M';
      EMPLEADO.SALARIO := '25000.00';
      EMPLEADO.NOMBREDEPTO := '';
      $STORE EMPLEADO;
```

El resultado de la orden STORE es que el contenido actual del registro UWA del tipo de registros especificado se inserta en la base de datos. Además, si el tipo de registros es miembro AUTOMATIC de un tipo de conjuntos, el registro se insertará automáticamente en un ejemplar de conjunto determinado por la declaración SET SELECTION. El registro recién insertado también se convertirá en el CRU y en el registro actual de su tipo de registros, así como en el actual de conjunto para cualquier tipo de conjuntos que tenga ese tipo de registros como propietario o como miembro.

Efectos de las opciones SET SELECTION sobre la orden STORE. Las opciones SET SELECTION AUTOMATIC tienen diferentes efectos sobre la ejecución de la orden STORE. Recuérdese que, en un tipo de conjuntos con opción de inserción AUTOMATIC, un registro nuevo del tipo de registros miembro se debe conectar a un ejemplar de conjunto en el momento en que se inserta en la base de datos mediante una orden STORE. Ahora haremos un breve examen de tres de las opciones SET SELECTION: STRUCTURAL, BY APPLICATION y BY VALUE.

En primer lugar, ilustraremos la opción **STRUCTURAL**. Recordemos, de la sección 10.3 (Fig. 10.10), que en el DDL de red esta opción tiene el siguiente formato:

> **SET SELECTION IS STRUCTURAL** <elemento de información> **IN** <tipo de registros miembro> = <elemento de información> **IN** <tipo de registros propietario>

Esto permite al SGBD determinar *por sí mismo* la ocurrencia de conjunto en la que se debe conectar un registro miembro recién insertado; se ilustra con las declaraciones de los tipos de conjuntos P_TRABAJAEN y E_TRABAJAEN de la figura 10.10(b). Por ejemplo, si queremos relacionar el registro EMPLEADO cuyo NSS es '567342739', recién insertado en EJ7, como trabajador de 40 horas por semana en el proyecto cuyo número es 55, tendremos que crear y almacenar un nuevo registro de enlace TRABAJA_EN con los valores apropiados de NSSE y NÚMEROP,

como se aprecia en EJ8. La orden STORE TRABAJA_EN de EJ8 conecta automáticamente el registro TRABAJA_EN recién insertado en el ejemplar de conjunto E_TRABAJAEN propiedad del registro EMPLEADO cuyo NSS es '567342739' y en el ejemplar de conjunto P_TRABAJAEN propiedad del registro PROYECTO cuyo NÚMERO es 55, mediante la localización automática de estos registros propietarios y sus ejemplares de conjunto. Además, el registro recién insertado se convierte en el actual de conjunto de estos dos tipos de conjuntos. Si no existiera alguno de los registros propietarios en la base de datos, la orden STORE generaría un error y el nuevo registro TRABAJA_EN no se insertaría en la base de datos.

```
EJ8:  TRABAJA_EN.NSSE := '567342739';
      TRABAJA_EN.NÚMEROP := 55;
      TRABAJA_EN.HORAS := '40.0';
      $STORE TRABAJA_EN;
```

En el DDL de red, la opción **BY APPLICATION** (por aplicación) tiene el siguiente formato:

> **SET SELECTION IS BY APPLICATION**

El programa de aplicación se encarga de seleccionar la ocurrencia de conjunto apropiada *antes* de almacenar el nuevo registro miembro. Por ejemplo, si queremos insertar un nuevo registro PROYECTO para un proyecto controlado por el departamento de investigación, deberemos establecer explícitamente el registro DEPARTAMENTO de 'Investigación' como actual de conjunto para CONTROLA *antes* de emitir la orden STORE PROYECTO.

En el DDL de red, la opción **BY VALUE** (por valor) tiene el siguiente formato:

> **SET SELECTION IS BY VALUE OF** <elemento de información> **IN** <tipo de registros propietario>

Ésta se ilustra con la declaración del tipo de conjuntos ES_SUPERVISOR en la figura 10.10(b). En este caso simplemente estableceremos el valor del campo especificado en la declaración SET SELECTION IS BY VALUE —NSS de EMPLEADO en el ejemplar del conjunto ES_SUPERVISOR— antes de emitir la orden STORE. El campo en cuestión debe ser un campo clave del tipo de registros propietario, y el SGBD usará ese valor para buscar el propietario (único) del registro nuevo. Por ejemplo, si queremos insertar un nuevo registro SUPERVISOR que corresponda al empleado cuyo NSS es '567342739', usaremos EJ9, donde el valor de EMPLEADO.NSS se provee en el programa para que el SGBD lo use en la selección del registro propietario EMPLEADO apropiado y conecte el nuevo registro SUPERVISOR a su ejemplar de conjunto.

Adviértase que podríamos haber declarado SET SELECTION IS STRUCTURAL para el tipo de conjuntos ES_SUPERVISOR; de hecho, esto habría sido más apropiado en la figura 10.10(b). No obstante, si el campo NSS_SUPERVISOR no estuviera incluido en el tipo de registros SUPERVISOR, no podríamos usar la opción STRUCTURAL y la opción más apropiada habría sido BY VALUE. En general, cuando un valor de campo único del tipo de registros propietario se duplica en el tipo de registros miembro, lo más apropiado es especificar SET SELECTION IS STRUCTURAL para los conjuntos automáticos; en los demás casos SET SELECTION debe ser BY VALUE o bien BY APPLICATION.

```
EJ9:  SUPERVISOR.NSS_SUPERVISOR := '567342739';
      (* crear nuevo registro SUPERVISOR en el UWA *)
      EMPLEADO.NSS := '567342739';
      (* fijar VALUE de NSS para selección automática de conjunto *)
      $STORE SUPERVISOR;
```

Las órdenes ERASE y ERASE ALL. A continuación analizaremos la eliminación de registros. Si queremos eliminar un registro de la base de datos, primero lo convertimos en el CRU y luego emitimos la orden **ERASE** (borrar). Por ejemplo, si queremos eliminar el registro EMPLEADO que insertamos en EJ7, podemos usar EJ10:

```
EJ10:  EMPLEADO.NSS := '567342739';
       $FIND ANY EMPLEADO USING NSS;
       if DB_STATUS = 0 then $ERASE EMPLEADO;
```

El efecto de una orden ERASE sobre cualesquier registros miembro que sean *propiedad del registro que se elimina* lo determina la opción de retención en conjuntos. Por ejemplo, el efecto de la orden ERASE en EJ10 depende de la retención de todos los tipos de conjuntos que tengan a EMPLEADO como propietario. Si la retención es OPTIONAL (opcional), los registros miembro se conservan en la base de datos pero se desconectan del registro propietario antes de que éste sea eliminado. Si la retención es FIXED (fija), todos los registros miembro se eliminan junto con su propietario. Por último, si la retención es MANDATORY (obligatoria) y el registro por eliminar es propietario de algún registro miembro, se rechaza la orden ERASE y se genera un mensaje de error. No es posible eliminar el propietario, porque si lo hiciéramos los registros miembro no tendrían propietario, lo cual está prohibido en los conjuntos MANDATORY. Estas reglas se aplican de manera recursiva a cualesquier registros adicionales que sean propiedad de otros registros cuya eliminación se efectúe automáticamente debido a una orden ERASE. Así, la eliminación puede propagarse en toda la base de datos y ser muy dañina si no se maneja con cuidado.

En EJ10, cuando borramos el registro EMPLEADO, se eliminan automáticamente todos los registros TRABAJA_EN y DEPENDIENTE de su propiedad, porque los conjuntos E_TRABAJAEN y DEPENDIENTES_DE tienen retención FIXED. Sin embargo, si ese registro EMPLEADO posee un registro SUPERVISOR a través del conjunto ES_SUPERVISOR o un registro DEPARTAMENTO a través del conjunto DIRIGE, el sistema rechaza la eliminación porque los conjuntos ES_SUPERVISOR y DIRIGE poseen retención obligatoria. Primero debemos eliminar explícitamente esos registros miembro de tales conjuntos MANDATORY antes de emitir la orden ERASE sobre su registro propietario. Si el registro EMPLEADO no posee registros SUPERVISOR ni DEPARTAMENTO a través de ES_SUPERVISOR o DIRIGE, el registro EMPLEADO se elimina.

Una variante de la orden ERASE, **ERASE ALL** (borrar todo), permite al programador eliminar un registro y todos los registros de los cuales es propietario directa o indirectamente. Esto significa que se borrarán *todos* los registros miembro propiedad de ese registro y también los registros miembro propiedad de cualquiera de los registros eliminados, y así sucesivamente. Por ejemplo, EJ11 elimina el registro DEPARTAMENTO de 'Investigación', así como todos los registros EMPLEADO propiedad de ese departamento a través de PERTENECE_A y todos los registros PROYECTO propiedad de ese DEPARTAMENTO a través de CONTROLA. Además, se eliminarán *automáticamente* cualesquier registros DEPENDIENTE, SUPERVISOR, DEPARTAMENTO o TRABAJA_EN que sean propiedad de los registros EMPLEADO o PROYECTO eliminados:

```
EJ11:  DEPARTAMENTO.NOMBRE := 'Investigación';
       $FIND ANY DEPARTAMENTO USING NOMBRE;
       if DB_STATUS = 0 then $ERASE ALL DEPARTAMENTO;
```

También podemos eliminar varios registros con un programa cíclico. Por ejemplo, suponga que deseamos eliminar todos los empleados que pertenecen al departamento de investigación, pero no el registro DEPARTAMENTO mismo; para ello usamos EJ12. Adviértase que

el CRU y el actual del tipo de registros del registro recién eliminado apuntan a una *posición "vacía"* donde estaba el registro que se eliminó. Gracias a esto, la orden FIND NEXT de EJ12 funciona correctamente:

```
EJ12:  DEPARTAMENTO.NOMBRE := 'Investigación';
       $FIND ANY DEPARTAMENTO USING NOMBRE;
       if DB_STATUS = 0 then
           begin
           $FIND FIRST EMPLEADO WITHIN PERTENECE_A;
           while DB_STATUS = 0 do
               begin
               $ERASE EMPLEADO;
               $FIND NEXT EMPLEADO WITHIN PERTENECE_A
               end
           end;
```

La orden MODIFY. La última orden para actualizar registros es **MODIFY**, que cambia los valores de algunos campos de un registro. Para modificar los valores de campos de un registro debemos seguir estos pasos:

- Hacer que el registro por modificar sea el CRU.
- Obtener el registro y colocarlo en la variable UWA correspondiente.
- Modificar los campos apropiados de la variable UWA.
- Emitir la orden MODIFY.

Por ejemplo, si queremos conceder a todos los empleados del departamento de investigación un aumento de sueldo del 10%, podemos usar EJ13. Suponemos la existencia de dos funciones en PASCAL —convertir_en_real y convertir_en_cadena— que se declararon en otra parte del programa; la primera convierte un valor de cadena del campo SALARIO en un número real, y la segunda da el formato de cadena del campo SALARIO a un valor real.

```
EJ13:  DEPARTAMENTO.NOMBRE := 'Investigación';
       $FIND ANY DEPARTAMENTO USING NOMBRE;
       if DB_STATUS = 0 then
           begin
           $FIND FIRST EMPLEADO WITHIN PERTENECE_A;
           while DB_STATUS = 0 do
               begin
               $GET EMPLEADO;
               EMPLEADO.SALARIO := convertir_en_cadena (convertir_en_real
                                   (EMPLEADO.SALARIO)*1.1);
               $MODIFY EMPLEADO;
               $FIND NEXT EMPLEADO WITHIN PERTENECE_A
               end
           end;
```

Órdenes para actualizar ejemplares de conjuntos. Ahora consideraremos las tres operaciones de actualización de conjuntos —CONNECT, DISCONNECT y RECONNECT— que sirven para insertar y eliminar registros miembro en ejemplares de conjuntos. La orden CONNECT

(conectar) inserta un registro miembro en un ejemplar de conjunto. El registro miembro debe ser el actual de unidad de ejecución y se conectará al ejemplar de conjunto que sea el actual de conjunto del tipo en cuestión. Por ejemplo, si queremos conectar el registro EMPLEADO cuyo NSS es '567342739' al conjunto PERTENECE_A propiedad del registro DEPARTAMENTO de 'Investigación', podemos usar EJ14:

```
EJ14:  DEPARTAMENTO.NOMBRE := 'Investigación';
       $FIND ANY DEPARTAMENTO USING NOMBRE;
       if DB_STATUS = 0 then
           begin
           EMPLEADO.NSS := '567342739';
           $FIND ANY EMPLEADO USING NSS;
           if DB_STATUS = 0 then
               $CONNECT EMPLEADO TO PERTENECE_A;
           end;
```

En EJ14, primero localizamos el registro DEPARTAMENTO de 'Investigación' para que el actual de conjunto del tipo PERTENECE_A se convierta en el ejemplar de conjunto propiedad de ese registro. Después localizamos el registro EMPLEADO requerido para que se convierta en el CRU. Por último, emitimos una orden CONNECT. Observe que el registro EMPLEADO que ha de conectarse *no debe ser miembro* de ningún ejemplar de conjunto de PERTENECE_A antes de que se emita la orden CONNECT. Si es esta la situación, deberemos usar la orden RECONNECT. La orden CONNECT sólo puede usarse con conjuntos MANUAL o AUTOMATIC OPTIONAL. En el caso de otros conjuntos AUTOMATIC, el sistema conecta automáticamente un registro miembro a un ejemplar de conjunto, gobernado por la opción SET SELECTION especificada, tan pronto como se haya almacenado el registro.

La orden DISCONNECT (desconectar) sirve para eliminar un registro miembro de un ejemplar de conjunto sin conectarlo a otro ejemplar. Por ello, sólo puede usarse con conjuntos OPTIONAL. Antes de emitir la orden DISCONNECT, hacemos que el registro por desconectar sea el CRU. Por ejemplo, si queremos eliminar el registro EMPLEADO con NSS = '83648 3873' del ejemplar de conjunto SUPERVISADOS del cual es miembro, empleamos EJ15:

```
EJ15:  EMPLEADO.NSS := '836483873';
       $FIND ANY EMPLEADO USING NSS;
       if DB_STATUS = 0
           then $DISCONNECT EMPLEADO FROM SUPERVISADOS;
```

Para concluir, la orden **RECONNECT** (reconectar) se puede usar con los conjuntos OPTIONAL y MANDATORY, pero no con los FIXED. Esta orden pasa un registro miembro de un ejemplar de conjunto a otro del *mismo* tipo de conjuntos. No se puede usar con conjuntos FIXED porque esta restricción prohíbe pasar un registro miembro de un ejemplar de conjunto a otro. Antes de emitir la orden RECONNECT el ejemplar de conjunto al cual se conectará el registro miembro deberá ser el actual de ese tipo de conjuntos, y el registro miembro que se ha de conectar deberá ser el CRU. Para lograr esto, necesitamos usar una frase adicional con la orden FIND —la frase RETAINING CURRENCY— que explicaremos en seguida.

La frase RETAINING CURRENCY. La orden RECONNECT y la frase **RETAINING CURRENCY** (reteniendo actualidad) se ilustran en el contexto de EJ16, que quita al gerente actual del departamento de investigación y asigna al empleado con NSS = '836483873' como nuevo gerente. Observe que el conjunto DIRIGE se declaró como AUTOMATIC MANDATORY, así que

otro registro EMPLEADO es actualmente propietario del registro DEPARTAMENTO de 'Investigación' en el tipo de conjuntos DIRIGE. Antes de emitir la orden RECONNECT, debemos hacer que el registro EMPLEADO con NSS = '836483873' sea el actual de conjunto del tipo de conjuntos DIRIGE. También debemos hacer que el registro DEPARTAMENTO de 'Investigación' sea el CRU, pues éste es el registro que ha de reconectarse a su nuevo gerente dentro del conjunto DIRIGE. Sin embargo, el registro DEPARTAMENTO de 'Investigación' ya es miembro de un ejemplar de conjunto DIRIGE distinto, de modo que cuando se le convierta en el CRU también se convertirá en el actual de conjunto del tipo de conjuntos DIRIGE:

```
EJ16:  EMPLEADO.NSS := '836483873';
       $FIND ANY EMPLEADO USING NSS; (* establecer actual de conjunto
       para DIRIGE *)
       if DB_STATUS = 0 then
           begin
           DEPARTAMENTO.NOMBRE = 'Investigación';
           $FIND ANY DEPARTAMENTO USING NOMBRE RETAINING DIRIGE
           CURRENCY;
           (* establecer el CRU sin cambiar el actual de conjunto de DIRIGE *)
           if DB_STATUS = 0 then $RECONNECT DEPARTAMENTO WITHIN
           DIRIGE
           end;
```

Para que el registro se convierta en el CRU *sin alterar el actual de conjunto*, agregamos la frase RETAINING CURRENCY al final de la orden FIND. En EJ16 anexamos a la orden FIND la frase RETAINING DIRIGE CURRENCY. Esto cambia el CRU al registro DEPARTAMENTO de 'Investigación' pero no modifica el actual de conjunto de DIRIGE; sigue siendo el ejemplar de conjunto propiedad del registro EMPLEADO cuyo NSS es '836483873'.

Cabe señalar que el registro por reconectar debe ser miembro de un ejemplar de conjunto del mismo tipo de conjuntos; de lo contrario tendríamos que usar la orden CONNECT. En el caso de conjuntos OPTIONAL se puede reemplazar un RECONNECT por un DISCONNECT seguido de un CONNECT. No obstante, en el caso de los conjuntos MANDATORY es necesario usar la orden RECONNECT si queremos pasar un registro miembro de un ejemplar de conjunto a otro, porque el registro debe permanecer conectado todo el tiempo a un propietario.

10.6 Un sistema de bases de datos de red: IDMS★

10.6.1 *Introducción*

En esta sección estudiaremos un SGBD muy popular que se basa en el modelo de red: el Integrated Database Management System (IDMS: sistema de gestión de bases de datos integrado). En la actualidad es comercializado por Computer Associates con el nombre CA-IDMS. El modelo de red ha tenido varias implementaciones importantes lanzadas al mercado: IDS II de Honeywell, DMS II de Burroughs (ahora UNISYS), DMS 1100 de UNIVAC (ahora UNISYS), VAX-DBMS de Digital Equipment, IMAGE de Hewlett-Packard e IDMS. Casi todos estos sistemas (con la excepción del IDS original), fueron creados después de publicarse el informe CODASYL DBTG de 1971 (DBTG 1971) y pusieron en práctica los conceptos especificados en

ese informe. El informe DBTG fue enmendado en 1978 y en 1981, con la adición de algunos conceptos y la eliminación de otros. Por ejemplo, en el informe de 1978 se agregaron las descripciones de estructuras de datos o vistas en términos de múltiples tipos de registros (de lo cual no hemos hablado). Se desecharon el modo de localización (LOCATION MODE) para definir tipos de registros y el concepto AREA (que tampoco analizamos) del informe original. Cuando se estudie cualquier sistema CODASYL en particular (a veces denominado sistema DBTG), en general tendríamos que explorar la mayoría de las características del modelo de red; en caso contrario, la clasificación CODASYL DBTG del sistema se tornará dudosa. Existen variaciones secundarias, algunas de las cuales pueden atribuirse a las actualizaciones del informe DBTG de 1978 y 1981.

DMS II de UNISYS no es una implementación CODASYL DBTG verdadera, pues no se ajusta estrictamente al modelo de red. Maneja los tipos de registros incorporados y, por tanto, puede considerarse que es tanto jerárquico como de red. El sistema TOTAL de CINCOM *tampoco* se ajusta a los conceptos DBTG. Representa los datos en términos de dos tipos de registros, llamados *maestro* y *variable*. Se pueden definir vínculos entre cualquier tipo de registros maestro y cualquier tipo de registros variable, pero no entre tipos de registros variables. A partir de entonces TOTAL se ha mejorado para convertirlo en SUPRA.

El DBTG propuso tres lenguajes:

- DDL de esquema — para describir una base de datos estructurada de red. Esto equivale al esquema conceptual ANSI/SPARC (Sec. 2.2).
- DDL de subesquema — para describir la parte de la base de datos pertinente para una aplicación. Esto corresponde al esquema externo ANSI/SPARC.
- DML — lenguaje de manipulación de datos para procesar los datos definidos por los dos lenguajes anteriores. El DML de DBTG (1971) se propuso para emplearse en conjunción con COBOL y, por tanto, se le llamó COBOL DML.

Todas las implementaciones del modelo de red tienen su propia sintaxis para estos tres lenguajes. La sintaxis de DML que hemos adoptado en este capítulo es muy parecida a la de IDMS. Todos los sistemas emplean el concepto de áreas de trabajo del usuario (UWA) y el de indicadores de actualidad, como se aprecia en la figura 10.13. Algunos SGBD (incluido IDMS) también se valen de un lenguaje de control de medios de dispositivos (DMCL: *device media control language*) para definir las características físicas de los medios de almacenamiento, como por ejemplo los tamaños de almacenamiento intermedio (*buffers*) y de páginas en los cuales se transforma la definición del esquema.

IDMS es la implementación original de los conceptos del CODASYL DBTG que realizó Cullinet Software. Está diseñado para ejecutarse en computadores centrales IBM con todos los sistemas operativos estándar. El nombre del producto se cambió oficialmente a IDMS/R (IDMS/Relacional) en 1983, cuando se añadieron recursos relacionales al producto base. Actualmente se le conoce como CA-IDMS y tiene dos versiones: DB y DC. En esta sección nos concentraremos en los recursos básicos orientados a red que corresponden al IDMS original. IDMS está íntimamente integrado a un producto de diccionario denominado **Integrated Data Dictionary** (IDD).

10.6.2 *Arquitectura básica de IDMS*

La familia de productos IDMS provee diversos recursos basados en el SGBD central y en el IDD. Este último almacena diversas entidades (término de IDD). Entre las entidades básicas se

cuentan: usuarios, sistemas, archivos, elementos de información, informes, transacciones, programas y puntos de entrada de los programas. Las entidades de teleproceso abarcan mensajes, pantallas, formatos de presentación, colas, destinos, líneas, terminales y tablas de codificación. También se almacenan varios vínculos y referencias cruzadas entre dichas entidades.

Los recursos de definición de datos incluyen tres compiladores que compilan el DDL de esquema, el DDL de subesquema y el DMCL. IDMS se invoca mediante una interfaz CALL para manipulación de datos. Los usuarios no codifican las llamadas en sus programas (a diferencia de IMS); en vez de ello, usan un conjunto de órdenes de DML similares a los de la tabla 10.2. Un **preprocesador de DML** traduce las órdenes de DML a secuencias de llamada apropiadas para el lenguaje anfitrión. IDMS provee recursos de DML dentro de los siguientes lenguajes anfitriones: COBOL, PL/1 y lenguaje ensamblador de IBM. Las extensiones relacionales de IDMS/R incluyen el Automatic System Facility (ASF: recurso automático del sistema), un sistema de máquina frontal controlado por menús con el que es posible, mediante un conjunto de funciones basadas en formas, definir y manipular vistas relacionales como si fueran registros lógicos. El Logical Record Facility (LRF: recurso de registros lógicos) crea una vista de la base de datos subyacente en forma de tablas virtuales para su procesamiento relacional.

Computer Associates suministra un conjunto de productos de servicio para usarse con IDMS. Además de IDD, incluyen los siguientes:

- *Generador de aplicaciones* (Application Development System/On-line Application Generator — ADS): Con esta herramienta, un creador de aplicaciones define las funciones de la aplicación y las respuestas al diccionario. Actúa como herramienta de prototipos y permite al usuario visualizar en línea la aplicación que está creando.
- *Consulta en línea* (OLQ: *On-line query*): Esta interfaz permite a los usuarios hacer consultas *ad hoc* a la base de datos u obtener informes con formato mediante consultas predefinidas.
- *Elaborador de informes* (CULPRIT): Éste es un elaborador de informes controlado por parámetros. Utiliza activamente la definición almacenada en el diccionario para generar informes. Es posible almacenar definiciones de informes en el diccionario e invocarlas suministrando el nombre del informe y sus parámetros.

Otros productos se mencionan en diversos cuadros del entorno CA-IDMS (Fig. 10.15).

A partir del repertorio de recursos antes mencionado, la creación de una aplicación IDMS se efectúa como sigue:

1. Se definen el esquema de la base de datos, el subesquema, etc., mediante herramientas interactivas llamadas utilerías de IDD.
2. Se compilan los esquemas con el compilador de esquemas.
3. El compilador de DMCL compila una descripción DMCL que define las características físicas de la base de datos.
4. El compilador de subesquemas compila subesquemas para diversas aplicaciones. Los registros lógicos o vistas son parte del subesquema en IDMS/R.
5. Los programas de aplicación fuente se escriben en un lenguaje anfitrión con órdenes DML incorporadas y se precompilan. Los precompiladores registran en el diccionario las operaciones que cada programa efectúa sobre datos específicos. El IDD vigila automáticamente esta información, y por ello se le denomina *diccionario activo*.

10.6.3 *Definición de datos en* IDMS

El esquema se define mediante el DDL de esquema. El compilador de esquemas en línea permite usar una sintaxis sin formato, modificar los esquemas por incrementos y validar los esquemas. Todo esquema tiene cinco clases de descripción distintas: de esquema, de archivos, de áreas, de registros y de conjuntos. Nos concentraremos en las últimas dos. Las definiciones de tipos de registros y de conjuntos para definir el esquema de la figura 10.9, que se muestran en la figura 10.10, pueden utilizarse en IDMS con pequeñas variaciones de la sintaxis. No analizaremos la sintaxis completa; sólo destacaremos las diferencias entre el DDL de IDMS y el DDL de la sección 10.3. Sus diferencias principales son:

- Cuando se define un tipo de registros se le debe asignar a un área, con la frase "WITHIN AREA".
- Todo tipo de registros debe tener una especificación LOCATION MODE (modo de localización).

Un **área** es un concepto de DBTG que se refiere a un grupo de tipo de registros. Por lo regular, las áreas corresponden a un espacio de almacenamiento físicamente contiguo. Este concepto tiene implicaciones físicas y pone en entredicho la independencia con respecto a los datos, por lo que fue excluido del informe DBTG aparecido en 1981. El **modo de localización** para un tipo de registros es una especificación de cómo debe almacenarse una ocurrencia nueva de ese registro y cómo deben leerse las ocurrencias existentes. IDMS permite los siguientes modos de localización:

- CALC — El registro se almacena mediante una clave CALC que forma parte del registro; con esta clave se calcula una dirección de página por dispersión y el registro se almacena en esa página o cerca de ella. La clave CALC puede declararse como única (DUPLICATES NOT ALLOWED).
- VIA — El VIA seguido de un nombre de tipo de conjuntos significa que los registros miembro se almacenarán físicamente lo más cerca posible al registro propietario (si pertenecen a la misma área). Si se asignan a áreas distintas, el registro miembro se almacenará en la *misma posición relativa* en su área que el propietario en la suya. Esta característica fue válida hasta la versión 10.0 de IDMS.
- VIA INDEX — En esta opción, disponible después de la versión 10.0, los registros se almacenan a través de un "índice propiedad del sistema" que incluye un registro propietario del sistema y un índice de árbol B^+. El registro propietario del sistema contiene el nombre del índice y apunta al árbol B^+ que, a su vez, apunta a los registros miembro.
- DIRECT — Los registros se colocan en una página especificada por el usuario, o cerca de ella.

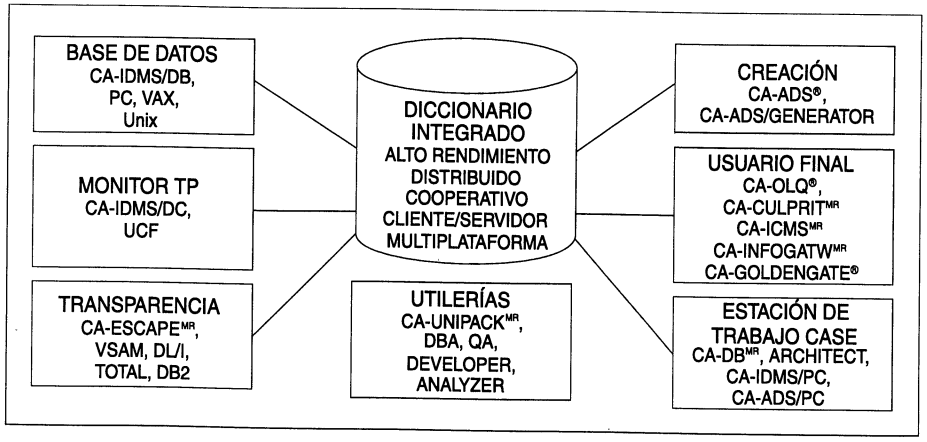


Figura 10.15 La familia de productos IDMS. (Cortesía de Computer Associates, *CA-IDMS/DB: Product Concepts and Facilities Manual*).

Existe también otra opción, llamada secuencial física. En la figura 10.16 presentamos muestras de definiciones para los tipos de registros EMPLEADO y TRABAJA_EN de la base de datos que vimos en la figura 10.9. Los elementos de información se definen según el estilo de COBOL. Observe la similitud entre esta definición y la de la figura 10.10. El acceso a EMPLEADO es a través de la clave CALC única NSS, y el de TRABAJA_EN es a través del tipo de conjuntos E_TRABAJAEN. Para que un usuario pueda comenzar una búsqueda en la base de datos directamente en un tipo de registros, éste debe haberse declarado con modo de localización

```
SCHEMA NAME IS COMPAÑÍA

RECORD NAME IS EMPLEADO
LOCATION MODE IS CALC USING NSS
DUPLICATES NOT ALLOWED
WITHIN ÁREA_EMP
02 NOMBREP  PIC X(15)
02 INIC  PIC X
02 APELLIDO PIC X(15)
02 NSS  PIC 9(9)
02 FECHANAC  PIC X(9)
02 DIRECCIÓN  PIC X(30)
02 SEXO  PIC X
02 SALARIO  PIC 9(10)
02 NOMBREDEPTO  PIC X(15)

RECORD NAME IS TRABAJA_EN
LOCATION MODE IS VIA E_TRABAJAEN SET
WITHIN ÁREA_EMP
02 NSSE  PIC 9(9)
02 NÚMEROP PIC 999 USAGE COMP-3
02 HORAS  PIC 99 USAGE COMP-3

SET NAME IS PERTENECE_A
OWNER IS DEPARTAMENTO
MEMBER IS EMPLEADO MANUAL OPTIONAL
ORDER IS SORTED
MODE IS CHAINED
ASCENDING KEY IS APELLIDO, NOMBREP
DUPLICATES ALLOWED
```

Figura 10.16 Muestras de definiciones de tipos de registros y de conjuntos en el DDL de esquema IDMS.

CALC o DIRECT. Otra opción consiste en utilizar un conjunto propiedad del sistema (Sec. 10.1.3) con ese tipo de registros como miembro.

IDMS también utiliza el concepto de **claves de base de datos**, que omitimos en nuestro análisis anterior. IDMS asigna un identificador único llamado clave de base de datos a cada ocurrencia de registro cuando se introduce en la base de datos. Su valor es un identificador de 4 bytes que contiene un número de página y un número de línea.

Definiciones de conjuntos. Las definiciones de conjuntos de IDMS difieren en sus características de las antes descritas en las siguientes formas:

1. Falta la cláusula SET SELECTION. Así, para insertar un registro automáticamente en el conjunto, el programa debe seleccionar una ocurrencia de conjunto apropiada, convirtiéndola en la actual.
2. No existe la retención en conjunto FIXED; sólo se permiten MANDATORY y OPTIONAL.
3. No hay recurso CHECK para verificar que un miembro de un conjunto satisfaga una cierta restricción antes de añadirse al conjunto (véase el ejemplo de CHECK con el conjunto PERTENECE_A en la figura 10.10(b)).
4. La definición de conjuntos incluye dos opciones de implementación: MODE IS CHAINED (el modo es encadenado) y MODE IS INDEXED (el modo es indizado). La primera vincula los miembros de toda ocurrencia de conjunto mediante una lista enlazada circular; la segunda establece un índice para cada ocurrencia de conjunto (véase la Sec. 10.1.4).
5. Es posible designar el orden en que se asignarán los apuntadores (al siguiente, al anterior, al propietario, etc.; véase la sección 10.1.4) dentro del registro. Éste es otro ejemplo de cómo se efectúa una especificación física de bajo nivel como parte del DDL en IDMS.

En la figura 10.16 mostramos cómo se definiría el tipo de conjuntos PERTENECE_A en el DDL.

Definición de subesquemas. Un subesquema en IDMS es un subconjunto del esquema original que se obtiene al omitir elementos de información, tipos de registros y tipos de conjuntos. Siempre que se omite un tipo de registros, deben eliminarse todos los tipos de conjuntos en los que participa como propietario o como miembro. La definición de subesquemas tiene dos divisiones de datos: división de identificación y división de datos de subesquema. La segunda especifica las áreas, los tipos de registros o partes de ellos y los conjuntos que se van a incluir. Se cuenta con un compilador de subesquemas en línea. La figura 10.17 muestra una definición de subesquema hipotética que contiene un subconjunto del esquema de la figura 10.9 relacionado con los empleados, los departamentos y los supervisores. La definición de subesquemas también puede incluir enunciados de registro lógico y grupo de trayectoria.

Cuando se aplica la orden ERASE a un registro del subesquema pueden surgir problemas. La eliminación puede propagarse a través de la pertenencia a conjuntos hasta varios otros registros que podrían no ser parte del subesquema. Al definir el subesquema, el diseñador debe incluir todos los tipos de registros a los que podrá propagarse la eliminación.

Descripción en el lenguaje de control de medios de dispositivos (DMCL). El DMCL permite especificar los parámetros de almacenamiento físico que gobiernan la correspondencia entre los datos y el almacenamiento para una descripción de esquema dada. Especifica el tamaño del almacenamiento intermedio (los *buffers*) en términos del número de páginas, y el tamaño de las páginas en bytes; asocia los nombres de área con los nombres de las reservas de almacenamiento intermedio, y da los nombres de los archivos de diario, especificando los tipos de dispositivos en los que dichos archivos residirán. No explicaremos aquí los detalles de la sintaxis del DMCL.

10.6.4 *Manipulación de datos en IDMS*

Los conceptos de manipulación de datos que presentamos en la sección 10.5, con algunas pequeñas modificaciones, son aplicables en IDMS. Todas las órdenes de DML de la tabla 10.2 están disponibles en alguna forma. A continuación veremos las variaciones.

Órdenes de obtención de datos

- FIND CALC <nombre-de-tipo-de-registros> es aplicable cuando ya se ha suministrado un valor clave en el campo de clave Calc del UWA.
- FIND <nombre-de-tipo-de-registros> DBKEY IS <valor-de-clave-de-bd> es una forma de buscar un registro, dado el valor de la clave de base de datos. Esta forma puede usarse sea o no DIRECT el modo de localización del registro, pero requiere que el programa proporcione el valor de la clave de base de datos (identificador de ubicación absoluta) de dicho registro. Esto se hace normalmente cuando un registro que ya se

```
ADD
        SUBSCHEMA NAME IS DEPTO_EMP OF SCHEMA NAME COMPAÑÍA
        DMCL NAME IS ED_DMCL
        PUBLIC ACCESS IS ALLOWED FOR DISPLAY
ADD
        AREA NAME IS ÁREA_EMP
        DEFAULT USAGE IS SHARED UPDATE
ADD
        AREA NAME IS ÁREA_DEP
ADD
        RECORD NAME IS EMPLEADO
                ELEMENTS ARE ALL
ADD
        RECORD NAME IS SUPERVISOR
                ELEMENTS ARE ALL
ADD
        RECORD NAME IS DEPARTAMENTO
                ELEMENTS ARE NÚMERO, NOMBRE
ADD
        SET NAME IS ES_SUPERVISOR
ADD
        SET NAME IS SUPERVISADOS
ADD
        SET NAME IS DIRIGE
ADD
        SET NAME IS PERTENECE_A
```

Figura 10.17 Definición de subesquema para el esquema de la figura 10.9 (se omiten los detalles).

leyó antes se debe leer otra vez en el programa; la clave de base de datos se guarda y se vuelve a utilizar.

- Para obtener un registro dentro de un tipo de conjuntos o dentro de un área, se dispone del siguiente FIND:

  FIND [FIRST | NEXT | PRIOR | LAST] <nombre-de-tipo-de-registros> [WITHIN <nombre-de-tipo-de-conjuntos> | WITHIN <nombre-de-área>]

  Los tipos de FIND disponibles se resumen en la figura 10.18. IDMS también permite usar el verbo OBTAIN (obtener) en vez de la combinación FIND-GET.

- El cuarto tipo de FIND de la figura 10.18 se puede usar no sólo dentro de un tipo de conjuntos, sino también dentro de un área.

Órdenes de actualización. Las órdenes STORE, ERASE y MODIFY de la sección 10.5.4 se aplican en IDMS. La orden ERASE (borrar) tiene cuatro opciones en IDMS:

- ERASE — Elimina el registro CRU (actual de unidad de ejecución) si no es el propietario de alguna ocurrencia de conjunto no vacía. El sistema tiene en cuenta todos los conjuntos en los cuales el tipo de registros es propietario.
- ERASE PERMANENT — Elimina el registro CRU, junto con las ocurrencias de miembros MANDATORY de ese registro en cualquier tipo de conjuntos. Las ocurrencias de miembros OPTIONAL no se eliminan pero sí se desconectan.
- ERASE SELECTIVE — Elimina el registro CRU, los miembros MANDATORY y los miembros OPTIONAL que no participan en ninguna otra ocurrencia de conjunto.
- ERASE ALL — Elimina el registro CRU y todos los miembros, sean MANDATORY u OPTIONAL.

En todas estas opciones, la eliminación se propaga recursivamente; esto es, como si el propio registro miembro eliminado fuera un objeto de la orden ERASE. Las órdenes CONNECT y DISCONNECT de IDMS funcionan tal como se explicó en la sección 10.5.4. No hay RECONNECT en IDMS.

10.6.5 *Almacenamiento de datos en IDMS*

En IDMS una base de datos se compone lógicamente de una o más áreas, las cuales a su vez se componen de páginas de la base de datos. Cada página corresponde a un bloque físico en

```
1 FIND ANY (o CALC)  <nombre-de-tipo-de-registros>
  FIND DUPLICATE     <nombre-de-tipo-de-registros>
2 FIND <nombre-de-tipo-de-registros> DBKEY IS <valor-de-clave-de-bd>
3 FIND CURRENT [ <nombre-de-tipo-de-registros> | WITHIN <nombre-de-tipo-de-
      conjuntos> | WITHIN <nombre-de-área> ]
4 FIND [ NEXT | PRIOR I FIRST | LAST | NTH ] <nombre-de-tipo-de-registros> [
      WITHIN <nombre-de-tipo-de-conjuntos> | WITHIN <nombre-de-área> ]
5 FIND <nombre-de-tipo-de-registros> WITHIN <nombre-de-tipo-de-conjuntos>
      USING <nombre-de-campo-de-ordenamiento>
```

Figura 10.18 Tipos de FIND disponibles en IDMS.

un archivo, por lo que la página es la unidad básica de entrada/salida. El vínculo entre áreas y archivos es de muchos a muchos; esto es, un área puede corresponder a varios archivos, y viceversa. Cabe destacar la similitud de este vínculo con los espacios en DB2 (y los grupos de almacenamiento en IMS). Esta correspondencia se almacena como parte de la descripción del esquema. Los archivos se dividen en bloques de longitud fija llamados páginas. Cada registro en una página tiene un prefijo que contiene un número de línea (contando desde la parte inferior de la página), un identificador de registro y una longitud de registro. El registro también tiene apuntadores: un mínimo de uno por cada tipo de conjuntos en el que es propietario o miembro. La clave de base de datos de un registro que está en la página 1051, línea 3, se considera como (1051,3).

Los conjuntos se implementan de dos maneras: como conjuntos enlazados (MODE IS CHAINED) o como conjuntos indizados (MODE IS INDEXED). Se incluye un apuntador hacia adelante para cada tipo de conjuntos en sus tipos de registros propietario y miembro; el diseñador también puede solicitar apuntadores hacia atrás (LINKED TO PRIOR, enlazado al previo), que se asignan tanto en el propietario como en el miembro, y un apuntador al propietario (LINKED TO OWNER, enlazado al propietario), que se asigna al registro miembro.

En la representación de conjunto indizado, cada ocurrencia de conjunto se representa con el propietario y un pequeño índice (local) representado por un conjunto de registros de índice. La figura 10.19 muestra el conjunto indizado PERTENECE_A para el esquema definido en la figura 10.16. Los registros propietario y de índice están vinculados en una lista enlazada mediante apuntadores al siguiente, al anterior y al propietario. Cada registro miembro apunta a su entrada de índice. *No es* necesario que un conjunto indizado esté ordenado.

Los conjuntos propiedad del sistema se mantienen como conjuntos indizados. Ahí la especificación de ordenamiento de conjuntos sirve para ordenar los valores del campo de ordenamiento. Cada registro de índice contiene este valor y el valor de la clave de base de datos. Hay una ocurrencia de cada conjunto, y el índice único así creado equivale a un índice de agrupamiento. No obstante, se pueden definir muchos conjuntos propiedad del sistema con diferentes ordenamientos de conjunto para el mismo tipo de registros. La opción CALC provee acceso disperso a un tipo de registros según una clave CALC (en IMS está disponible *sólo para los registros raíz*). Con la opción VIA SET (a través del conjunto), si ese conjunto se define con MODE IS INDEXED y ORDER IS SORTED, no sólo se almacenan los registros miembro cerca del propietario (en la misma página o en una cercana), sino que además el orden físico de los registros miembro es muy cercano a su orden lógico.

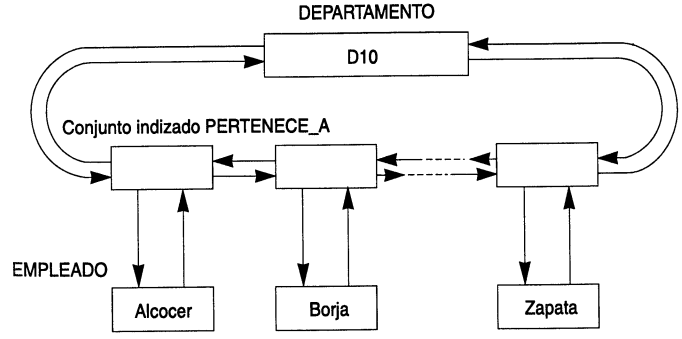


Figura 10.19 Una ocurrencia de un tipo de conjuntos indizado PERTENECE_A ordenado.

El producto encargado de procesar bases de datos centralizadas (sin telecomunicaciones) se llama CA-IDMS/DB. IDMS también está disponible, y proporciona un recurso de comunicaciones integrado en forma del producto CA-IDMS/DC. Los detalles rebasan el alcance de nuestro análisis, pues todavía no hemos hablado de las bases de datos distribuidas ni de las arquitecturas "cliente-servidor".

A fin de que el modelo relacional sea compatible con IDMS, éste se ha mejorado para manejar la coexistencia de procesamiento tanto de navegación como relacional, incluso dentro de la misma aplicación. Naturalmente, esto implica que los registros puedan ser tratados como filas de una tabla, donde el tipo de registros equivale a una tabla o definición de relación. Con los recursos de consulta y de programación es posible tener acceso a los datos en tres formas: mediante el DML de navegación; mediante el lenguaje SQL; y mediante SQL dinámico, en el que la consulta SQL se formula dentro de un programa en el momento de su ejecución. Estos mecanismos de múltiple acceso se manejan sobre el "nivel lógico". El "nivel físico" se ocupa de los datos almacenados en forma de archivos en BDAM (*Basic Direct Access Method:* método básico de acceso directo) y VSAM (*Virtual Sequential Access Method:* método de acceso secuencial virtual).

El SQL de CA-IDMS/DB se basa en la norma ANSI de SQL, nivel 2. El lenguaje contiene las opciones GRANT (otorgar) y REVOKE (revocar) (véase el Cap. 20), además del SQL estándar (analizado en el Cap. 7). El producto llamado CA Extended SQL cuenta con recursos más avanzados, como SQL dinámico y funciones de fecha/hora/subcadena en SQL. No nos ocuparemos a profundidad del procesamiento de bases de datos almacenadas bajo el modelo de red empleando el modelo relacional, pero es probable que los productos de SGBD de red actuales continúen con esa tendencia.

10.7 Resumen

En este capítulo estudiamos el modelo de red, en el que los datos se representan con tipos de registros y tipos de conjuntos como bloques de construcción. Cada tipo de conjuntos define un vínculo 1:*N* entre un tipo de registros propietario y un tipo de registros miembro. Un tipo de registros puede participar como propietario o miembro en cualquier cantidad de tipos de conjuntos. Ésta es la principal diferencia entre los tipos de conjuntos en el modelo de red y los vínculos padre-hijo del modelo jerárquico que veremos en el capítulo 11. En el modelo jerárquico, los vínculos deben obedecer un patrón estrictamente jerárquico. Como en la mayoría de los SGBD jerárquicos se aplican restricciones sobre los diferentes tipos de vínculos padre-hijo, el modelo de red posee mejores capacidades de modelado que el jerárquico. La capacidad de modelado que tiene el modelo de red también es superior a la del modelo relacional original en cuanto a que modela *explícitamente* los vínculos, aunque la incorporación de las claves externas en el modelo relacional subsana esta deficiencia. Las operaciones de reunión del modelo relacional se vuelven visibles y sustanciales como tipos de conjuntos en el modelo de red.

Vimos tres tipos de conjuntos especiales. Los conjuntos propiedad del sistema o singulares sirven para definir puntos de entrada a la base de datos. Los conjuntos multimiembro son útiles en los casos en que los registros miembro pueden ser de más de un tipo de registros. Los conjuntos recursivos son aquellos en los que el mismo tipo de registros participa como propietario y como miembro. Debido a problemas de implementación, los conjuntos recursivos estaban prohibidos en el modelo de red CODASYL original, pero ahora se acostumbra representarlos con un tipo de registros de enlace adicional y dos tipos de conjuntos.

Después analizamos los tipos de restricciones de integridad sobre la pertenencia a conjuntos que podemos especificar en un esquema de red. Éstos se clasifican como opciones de inserción (MANUAL o AUTOMATIC), opciones de retención (OPTIONAL, MANDATORY o FIXED) y opciones de ordenamiento.

Examinamos la representación de lista enlazada circular (o anillo) para implementar ejemplares de conjuntos, así como otras opciones de implementación con que se puede mejorar el rendimiento de la lista enlazada circular, como el doble enlace y los apuntadores al propietario. También mencionamos otras técnicas para implementar conjuntos en vez de listas enlazadas, como el almacenamiento contiguo o los arreglos de apuntadores.

Los vínculos M:*N*, es decir, los vínculos en los que pueden participar más de dos tipos de registros, también son representables con un tipo de registros de enlace. En el caso de un vínculo M:*N* con dos tipos de registros participantes, se usan dos tipos de conjuntos y un tipo de registros de enlace. En el caso de un vínculo *n*-ario en el que participan *n* tipos de registros, se necesita un tipo de registros de enlace y *n* tipos de conjuntos. Los vínculos 1:1 no se representan explícitamente; se pueden representar como un tipo de conjuntos, pero los programas de aplicación deben cerciorarse de que cada ejemplar de conjunto tenga cuando más un registro miembro en todo momento.

Se presentó un lenguaje de definición de datos (DDL) para el modelo de red. Vimos cómo se definen los tipos de registros y de conjuntos, y analizamos las diversas opciones SET SELECTION para los conjuntos AUTOMATIC. Estas opciones especifican la forma en que el SGBD identifica el ejemplar de conjunto apropiado de un tipo AUTOMATIC en el que habrá de conectarse un nuevo registro miembro cuando éste se almacene en la base de datos.

Presentamos las órdenes de un lenguaje de manipulación de datos (DML), que operan sobre un solo registro cada vez, para el modelo de red. Vimos cómo se escriben programas con órdenes de DML incorporadas para obtener información de una base de datos de red y también para actualizarla. La orden FIND sirve para "navegar" por la base de datos, estableciendo diversos indicadores de actualidad, y la orden GET se usa para obtener el CRU y colocarlo en la variable de programa UWA correspondiente. Así mismo, vimos órdenes para insertar, eliminar y modificar registros, y para modificar ejemplares de conjuntos.

Por último presentamos un panorama del SGBD de red IDMS que se encuentra actualmente en el mercado.

Preguntas de repaso

10.1. Analice los diversos tipos de campos (elementos de información) que se pueden definir para los tipos de registros en el modelo de red.

10.2. Defina los siguientes términos: *tipo de conjuntos, tipo de registros propietario, tipo de registros miembro, ejemplar (ocurrencia) de conjunto, tipo de conjuntos* AUTOMATIC, *tipo de conjuntos* MANUAL, *tipo de conjuntos* MANDATORY, *tipo de conjuntos* OPTIONAL, *tipo de conjuntos* FIXED.

10.3. ¿Cómo se identifican los ejemplares de conjunto de un tipo de conjuntos?

10.4. Explique las diversas restricciones sobre pertenencia a conjuntos y los casos en que debe usarse cada una de ellas.

10.5. En la representación de lista enlazada circular (anillo) de los ejemplares de conjunto, ¿cómo distingue el SGBD entre los registros miembro y el registro propietario de un ejemplar de conjunto?

10.6. Analice los diversos métodos para implementar ejemplares de conjuntos. Para cada método, indique qué tipos de órdenes FIND para procesar conjuntos se pueden implementar de manera eficiente y cuáles no.

10.7. ¿Para qué sirven los tipos de conjuntos propiedad del sistema (singulares)?

10.8. ¿Para qué se usan los tipos de conjuntos multimiembro?

10.9. ¿Qué son los tipos de conjuntos recursivos? ¿Por qué no se permiten en el modelo de red CODASYL original? ¿Cómo pueden implementarse mediante un tipo de registros de enlace?

10.10. Muestre cómo se representan en el modelo de red cada uno de los siguientes tipos de vínculos: (a) vínculos M:N; (b) vínculos n-arios con $n > 2$; (c) vínculos 1:1. Explique cómo puede transformarse un esquema ER a un esquema de red.

10.11. Analice los siguientes conceptos, y explique para qué sirve cada uno cuando se escribe un programa de base de datos en DML de red: (a) el área de trabajo del usuario (UWA); (b) los indicadores de actualidad; (c) el indicador de estado de la base de datos.

10.12. Analice los diferentes tipos de indicadores de actualidad, e indique de qué manera cada tipo de orden de navegación FIND afecta a cada uno de dichos indicadores.

10.13. Describa las diversas opciones SET SELECTION para tipos de conjuntos AUTOMATIC, y especifique las circunstancias en las que debe elegirse cada una de ellas.

10.14. Indique qué especifica cada una de las siguientes cláusulas del DDL de red: (a) DUPLICATES ARE NOT ALLOWED; (b) ORDER IS; (c) KEY IS; (d) CHECK.

10.15. ¿Para qué sirve la cláusula RETAINING CURRENCY cuando se usa con la orden FIND en el DML de red?

10.16. ¿Qué diferencias hay entre la orden ERASE ALL y la orden ERASE?

10.17. Analice las órdenes CONNECT, DISCONNECT y RECONNECT, y especifique los tipos de restricciones de conjunto bajo las cuales puede usarse cada una.

Ejercicios

10.18. Especifique las consultas del ejercicio 6.19 usando órdenes DML de red incorporadas en PASCAL aplicadas al esquema de base de datos de red que aparece en la figura 10.9. Utilice las variables de programa PASCAL declaradas en la figura 10.12, y declare cualesquier variables adicionales que pudiera necesitar.

10.19. ¿Cómo modificaría el segmento de programa de EJ3 para obtener, por *cada* departamento, el nombre del departamento y los nombres de todos los empleados que pertenecen a ese departamento, en orden alfabético?

10.20. Considere la siguiente consulta: "Para cada departamento, imprimir los nombres del departamento y de su gerente; para cada empleado que pertenezca a ese departamento, imprimir su nombre y la lista de nombres de los proyectos en los que trabaja." Escriba primero un segmento de programa para esta consulta, y luego modifíquelo de modo que se cumplan las siguientes condiciones, una por una:

a. Sólo aparecen en la lista los departamentos que tienen más de 10 empleados.

b. Sólo aparecen en la lista los empleados que trabajan más de 20 horas en total.

c. Sólo aparecen en la lista los empleados con dependientes.

10.21. Considere el esquema de base de datos de red que aparece en la figura 10.20, correspondiente al esquema relacional de la figura 2.1. Escriba enunciados apropiados en DDL de red para definir los tipos de registros y de conjuntos del esquema. Escoja restricciones de conjunto adecuadas para cada tipo de conjuntos, y justifique sus elecciones.

10.22. Escriba segmentos de programa en PASCAL con órdenes DML de red incorporadas para especificar las consultas del ejercicio 7.16 sobre el esquema de la figura 10.20.

10.23. Escriba segmentos de programa en PASCAL con órdenes DML de red incorporadas para especificar las consultas de los ejercicios 7.17 y 7.18 sobre el esquema de base de datos de red que aparece en la figura 10.20. Declare todas las variables de programa que necesite.

10.24. Durante el procesamiento de la consulta "buscar todos los cursos que ofrece el departamento CICO y, para cada curso, listar su nombre, sus secciones y sus requisitos previos" sobre el esquema de la figura 10.20, el SGBD efectúa las siguientes acciones: (a) buscar el registro DEPARTAMENTO de CICO; (b) buscar un registro CURSO para CICO541 como miembro de OFRECE; (c) buscar un registro SECCIÓN para la sección S32491 como miembro de TIENE_SECCIONES; (d) buscar una ocurrencia de registro REQUISITO —digamos, P1— como miembro de ES_CON_REQUISITO; (e) buscar un registro miembro CURSO de TIENE_REQUISITO —digamos, el curso MATE143—. Muestre los

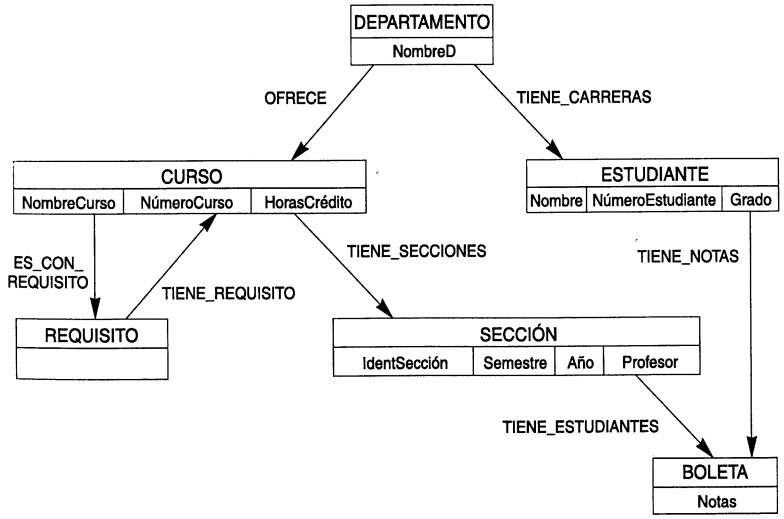


Figura 10.20 Esquema de red para una base de datos universitaria.

indicadores de actualidad de los tipos de conjuntos, de los tipos de registros y el CRU después de cada uno de estos sucesos, empleando la notación de la figura 10.14. Suponga que originalmente todos los indicadores de actualidad eran **nil**. ¿Qué sucede con el actual del tipo de registros CURSO después del paso (e)? Suponga que se ejecuta FIND NEXT CURSO WITHIN OFRECE después del paso (e); ¿qué problema se presentaría, y cómo podríamos resolverlo? (*Sugerencia:* Considere el empleo de la frase RETAINING CURRENCY en algunas de las órdenes.)

10.25. Escriba procedimientos en pseudocódigo (al estilo PASCAL) que puedan formar parte del software del SGBD, y describa a grandes rasgos las acciones realizadas por las siguientes órdenes de DML:

a. Procesar la orden STORE <tipo de registros>. Hay que comprobar que las definiciones de tipos de conjuntos están en el catálogo del SGBD para poder determinar en cuáles conjuntos participa el tipo de registros como propietario o miembro, y efectuar las acciones apropiadas.

b. Procesar las órdenes ERASE <tipo de registros> y ERASE ALL <tipo de registros>.

10.26. Escoja alguna aplicación de base de datos que conozca o que le interese.

a. Diseñe un esquema de base de datos de red para su aplicación.

b. Declare sus tipos de registros y de conjuntos mediante el DML de red.

c. Especifique varias consultas y actualizaciones que necesite su aplicación de base de datos, y escriba un segmento de programa en PÀSCAL con órdenes DML de red incorporadas para cada una de sus consultas.

d. Implemente su base de datos si dispone de un SGBD de red.

10.27. Establezca una transformación de los siguientes esquemas ER a esquemas de red. Especifique para cada tipo de conjuntos las opciones de inserción y retención y cualesquier opciones de ordenamiento; justifique sus decisiones.

a. El esquema ER AEROLÍNEA de la figura 3.19.

b. El esquema ER BANCO de la figura 3.20.

c. El esquema ER CONTROL_BUQUES de la figura 6.21.

10.28. Considere el diagrama de esquema de red para una base de datos BIBLIOTECA que se muestra en la figura 10.21, correspondiente al esquema relacional de la figura 6.22.

a. Especifique variables UWA de PASCAL apropiadas para dicho esquema.

b. Escriba segmentos de programa en PASCAL con órdenes DML de red incorporadas para cada una de las consultas del ejercicio 6.26.

c. Compare los esquemas relacional (Fig. 6.22) y de red (Fig. 10.21) para la base de datos BIBLIOTECA. Identifique sus similitudes y sus diferencias. ¿Cómo puede hacer que el esquema de red se parezca más al esquema relacional?

10.29. Trate de transformar el esquema de red de la figura 10.21 a un esquema ER. Esto es parte de un proceso llamado *ingeniería inversa*, en el cual se crea un esquema conceptual a partir de una base de datos ya implementada. Exprese todas las suposiciones que haga.

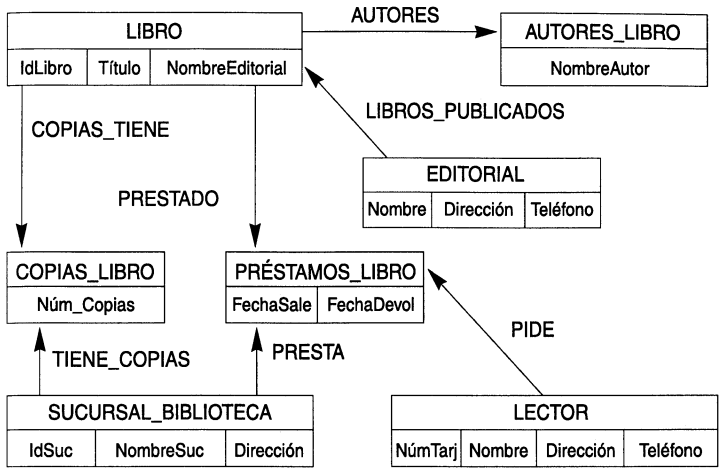


Figura 10.21 Diagrama de esquema de red para una base de datos BIBLIOTECA.

Bibliografía selecta

Los primeros trabajos sobre el modelo de datos de red los realizó Charles Bachman cuando se ocupaba en la creación del primer SGBD comercial, IDS (Bachman y Williams 1964), en General Electric, y después en Honeywell. Bachman también dio a conocer la primera técnica de diagramación para representar vínculos en los esquemas de bases de datos, los llamados diagramas de estructuras de datos (Bachman 1969) o diagramas de Bachman. Por sus trabajos Bachman obtuvo el Premio Turing de 1973, el más alto honor otorgado por la ACM, y en su conferencia para la ceremonia de entrega de dicha presea (Bachman 1973) expuso su visión sobre la base de datos como recurso primario y sobre el programador como "navegante" por la base de datos. En un debate entre partidarios y oponentes del enfoque relacional efectuado en 1974, se puso del lado de los segundos (Bachman 1974). Otros trabajos sobre el modelo de red son los de George Dodd (Dodd 1966) en General Motors Research. Dodd (1969) ofrece uno de los primeros estudios comprensivos en torno a las técnicas de gestión de bases de datos.

El DBTG (*Data Base Task Group*: Grupo de trabajo de bases de datos) de CODASYL (*Conference on Data Systems Languages*: Conferencia sobre lenguajes de sistemas de datos) se formó con el fin de proponer normas para los SGBD. El informe DBTG de 1971 (DBTG 1971) contiene lenguajes de definición de datos de esquema y de subesquema, así como un DML para usarse con COBOL. En 1978 se preparó un informe enmendado (CODASYL 1978), y en 1981 se hizo otra revisión en borrador. El comité X3H2 de ANSI (American National Standards Institute: Instituto nacional estadounidense de normas) propuso un lenguaje de red estándar, llamado NDL.

El diseño de las bases de datos de red se analiza en Dahl y Bubenko (1982), Whang *et al.* (1982), Schenk (1974), Gerritsen (1975) y Bubenko *et al.* (1976). Irani *et al.* (1979) examinan técnicas de optimización para diseñar esquemas de red a partir de los requerimientos del usuario. Bradley (1978) propone un lenguaje de consulta de alto nivel para el modelo de red. Navathe (1980) analiza la transformación estructural de esquemas de red a esquemas relacionales. Mark *et al.* (1992) estudia una estrategia para mantener una base de datos de red y relacional en un estado consistente.

Taylor y Frank (1976) presentan un panorama sobre la gestión de bases de datos de red. Los libros de Cardenas (1985), Kroenke y Dolan (1988) y Olle (1978) ofrecen tratamientos amplios del modelo de red. El sistema IDMS se describe en varios manuales de CA-IDMS/DB publicados por Computer Associates.

CAPÍTULO 11

El modelo de datos jerárquico y el sistema IMS

El modelo jerárquico de los datos se creó con el fin de modelar la gran cantidad de tipos de organizaciones jerárquicas que existen en el mundo real. Desde hace mucho tiempo los seres humanos han organizado la información en jerarquías para entender mejor el mundo. Hay muchos ejemplos, como los esquemas de clasificación para las especies de los reinos animal y vegetal, y las clasificaciones de los lenguajes humanos. El hombre también adoptó estructuras y esquemas de nomenclatura jerárquicos para las estructuras que él mismo iba creando, como los organigramas corporativos, los esquemas de clasificación bibliográficos y las jerarquías gubernamentales. El modelo de datos jerárquico representa organizaciones jerárquicas en forma directa y natural y puede ser la mejor opción en algunas situaciones, aunque presentará problemas cuando represente situaciones con vínculos que no sean jerárquicos.

No existe ningún documento original que describa el modelo jerárquico, como ocurre con los modelos relacional y de red. Más bien, varios de los primeros sistemas de gestión de información por computador se crearon empleando estructuras de almacenamiento jerárquicas. Hay ejemplos recientes de estos sistemas, como el Multi-Access Retrieval System (MARS VI) de Control Data Corporation, el Information Management System (IMS) de IBM y el System-2000 de MRI (ahora comercializado por SAS Institute).

En este capítulo estudiaremos los principios en que se basa el modelo jerárquico, independientemente de cualquier sistema específico, y los relacionaremos con IMS, que es el sistema jerárquico que más se usa en la actualidad. En la sección 11.1 examinaremos los esquemas y ejemplares jerárquicos. En la sección 11.2 veremos el concepto de vínculo padre-hijo virtual, con el que es posible compensar las limitaciones de las jerarquías puras. En la sección 11.3 trataremos las restricciones en el modelo jerárquico. En la sección 11.5

abordaremos la transformación de esquemas del modelo ER al modelo jerárquico. Las secciones 11.4 y 11.6 se ocuparán de los lenguajes de definición y de manipulación de datos para el modelo jerárquico de los datos tal como se define aquí. En la sección 11.7 analizaremos un sistema jerárquico representativo, IMS, mencionando también algunas de las características de otro sistema jerárquico: System 2000. La sección 11.8 resume este capítulo.

Los lectores que sólo deseen un breve panorama del modelo jerárquico pueden omitir lectura de algunas partes de las secciones 11.4 a 11.7, o todas.

11.1 Estructuras de bases de datos jerárquicas

En esta sección analizaremos los conceptos de estructuración de datos en el modelo jerárquico. En primer lugar hablaremos de los vínculos padre-hijo y de cómo podemos usarlos para formar un esquema jerárquico (en la Sec. 11.1.1). Luego estudiaremos las propiedades que tiene un esquema jerárquico (en la Sec. 11.1.2). Por último, examinaremos los árboles de ocurrencia jerárquicos en la sección 11.1.3, así como un método común para almacenarlos: la secuencia jerárquica (en la Sec. 11.1.4).

11.1.1 *Vínculos padre-hijo y esquemas jerárquicos*

El modelo jerárquico utiliza dos conceptos principales de estructuración de datos: registros y vínculos padre-hijo. Un **registro** es una colección de **valores de campos** que proporcionan información sobre una entidad o un ejemplar de vínculo. Los registros del mismo tipo se agrupan en **tipos de registros**. Cada tipo de registros recibe un nombre, y su estructura se define en términos de una colección de **campos** o **elementos de información** con nombre. Cada campo tiene un cierto tipo de datos, como entero, real o cadena.

Un **tipo de vínculos padre-hijo (tipo de VPH)** es un vínculo 1:*N* entre dos tipos de registros. El tipo de registros del lado 1 se denomina **tipo de registros padre**, y el del lado *N* se llama **tipo de registros hijo** del tipo de VPH. Una **ocurrencia** (o **ejemplar**) **del tipo de VPH** consiste en *un registro* del tipo de registros padre y *varios registros* (cero o más) del tipo de registros hijo.

Un **esquema de base de datos jerárquica** consiste en varios esquemas jerárquicos. Cada **esquema jerárquico** (o **jerarquía**) comprende varios tipos de registros y tipos de VPH.

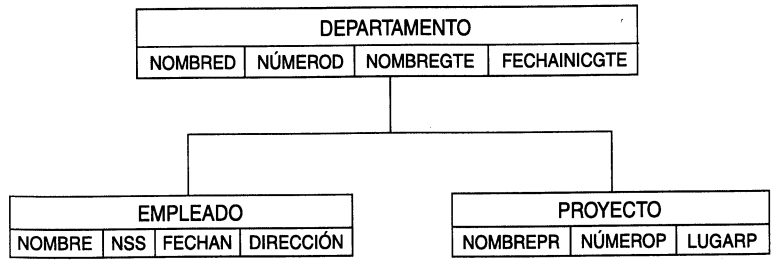


Figura 11.1 Un esquema jerárquico.

Los esquemas jerárquicos se visualizan como **diagramas jerárquicos**, en los cuales los nombres de los tipos de registros aparecen en cuadros rectangulares y los tipos de VPH se dibujan como líneas que conectan el tipo de registros padre y el tipo de registros hijo. En la figura 11.1 aparece un diagrama jerárquico simple de un esquema jerárquico que tiene tres tipos de registros y dos tipos de VPH. Los tipos de registros son DEPARTAMENTO, EMPLEADO y PROYECTO. Podemos exhibir los nombres de los campos debajo de cada nombre de tipo de registros, como se aprecia en la figura 11.1. En otros diagramas, por brevedad, sólo mostraremos los nombres de los tipos de registros.

En los esquemas jerárquicos haremos referencia a un tipo de VPH con el par (tipo de registros padre, tipo de registros hijo) entre paréntesis. Los dos tipos de VPH de la figura 11.1 son (DEPARTAMENTO, EMPLEADO) y (DEPARTAMENTO, PROYECTO). Observe que los tipos de VPH *no tienen* nombre en el modelo jerárquico. Sin embargo, el diseñador de la base de datos asocia un cierto significado a cada tipo de VPH. En la figura 11.1, cada *ocurrencia* del tipo de VPH (DEPARTAMENTO, EMPLEADO) relaciona un registro de departamento con los registros de los *múltiples* (cero o más) empleados que pertenecen a ese departamento. Una *ocurrencia* del tipo de VPH (DEPARTAMENTO, PROYECTO) relaciona un registro de departamento con los registros de los proyectos controlados por ese departamento. La figura 11.2 muestra dos ocurrencias (o ejemplares) de VPH para cada uno de estos dos tipos de VPH.

11.1.2 *Propiedades de los esquemas jerárquicos*

Un esquema jerárquico de tipos de registros y tipos de VPH debe tener las siguientes propiedades:

1. Un tipo de registros, la **raíz** del esquema jerárquico, no participa como tipo de registros hijo en ningún tipo de VPH.
2. Todo tipo de registros, con excepción de la raíz, participa como tipo de registros hijo en *uno y sólo un* tipo de VPH.
3. Un tipo de registros puede participar como tipo de registros padre en cualquier cantidad (cero o más) de tipos de VPH.

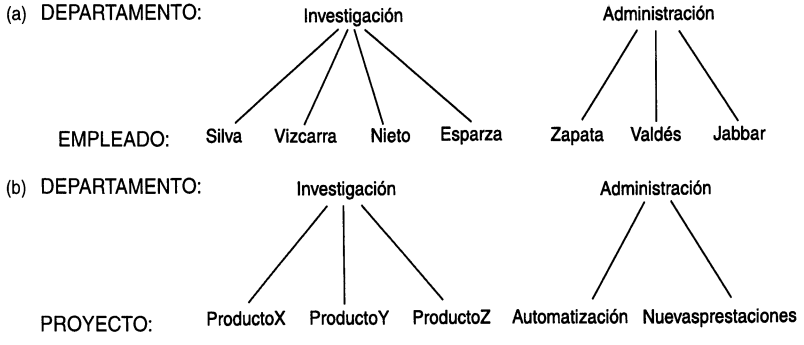


Figura 11.2 Ocurrencias de tipos de VPH. (a) Dos ocurrencias del tipo de VPH (DEPARTAMENTO, EMPLEADO). (b) Dos ocurrencias del tipo de VPH (DEPARTAMENTO, PROYECTO).

4. Un tipo de registros que no participa como tipo de registros padre en ningún tipo de VPH se denomina **hoja** del esquema jerárquico.
5. Si un tipo de registros participa como padre en más de un tipo de VPH, entonces *sus tipos de registros hijo están ordenados* El orden se visualiza, por convención, de izquierda a derecha en los diagramas jerárquicos.

La definición de esquema jerárquico define una **estructura de datos de árbol**. En la terminología de estas estructuras, un tipo de registros corresponde a un **nodo** del árbol, y un tipo de VPH corresponde a una **arista** (o **arco**) del árbol. Usaremos los términos *nodo* y *tipo de registros*, y *arista* y *tipo de VPH* indistintamente. La convención habitual para representar los árboles es un poco diferente de la que se adopta en los diagramas jerárquicos, en cuanto a que cada arista del árbol se muestra completamente separada de las demás (Fig. 11.3). En los diagramas jerárquicos la convención es que todas las aristas que emanan del mismo nodo padre se reúnen (Fig. 11.1). Adoptaremos esta última convención en los diagramas jerárquicos.

Las propiedades anteriores de los esquemas jerárquicos significan que todos los nodos, excepto la raíz, tienen uno y sólo un nodo padre. No obstante, un nodo puede tener varios nodos hijo, y en tal caso están ordenados de izquierda a derecha. En la figura 11.1 EMPLEADO es el primer hijo de DEPARTAMENTO y PROYECTO es el segundo. Las propiedades antes identificadas también limitan los tipos de vínculos que se pueden representar en un esquema jerárquico. En particular, los vínculos M:N *no pueden* representarse directamente porque los vínculos padre-hijo son 1:N, y un tipo de registros *no puede participar como hijo* en dos o más vínculos padre-hijo distintos.

En el modelo jerárquico es posible manejar los vínculos M:N *si se pueden duplicar los ejemplares de registros hijo*. Por ejemplo, consideremos un vínculo M:N entre EMPLEADO y PROYECTO, donde un proyecto puede tener varios empleados que trabajan en él, y un empleado puede trabajar en varios proyectos. Podemos representar el vínculo como un tipo

EJEMPLO 1: Consideremos los siguientes ejemplares del vínculo EMPLEADO:PROYECTO:

Proyecto	Empleados que trabajan en el proyecto
A	E1, E3, E5
B	E2, E4, E6
C	E1, E4
D	E2, E3, E4, E5

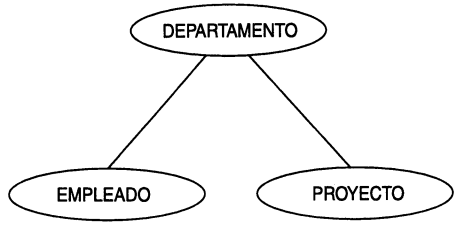


Figura 11.3 Representación de árbol del esquema jerárquico de la figura 11.1.

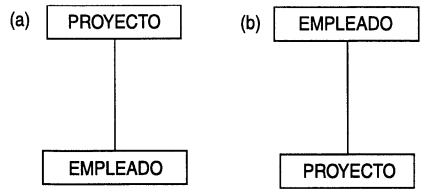


Figura 11.4 Representación de un vínculo M:N. (a) Una representación del vínculo M:N. (b) Representación alternativa del vínculo M:N.

de VPH (PROYECTO, EMPLEADO) como se muestra en la figura 11.4(a). En este caso, un registro que describa al mismo empleado puede duplicarse si aparece una vez debajo de *cada uno* de los proyectos en los que ese empleado trabaje. Como alternativa, podemos representar el vínculo como un tipo de VPH (EMPLEADO, PROYECTO) como se muestra en la figura 11.4(b), en cuyo caso podremos duplicar los registros de proyectos.

Si estos ejemplares se almacenan empleando el esquema jerárquico de la figura 11.4(a), habrá cuatro ocurrencias del tipo de VPH (PROYECTO, EMPLEADO): una por cada proyecto. En cambio, los registros de los empleados E1, E2, E3 y E5 aparecerán *dos veces cada uno* como registros hijo, porque cada uno de estos empleados trabajan en dos proyectos. El registro del empleado E4 aparecerá tres veces: una vez por cada uno de los proyectos B, C y D. Algunos de los valores de los campos de datos en los registros de empleados pueden depender del contexto; es decir, dichos valores dependerían tanto de EMPLEADO como de PROYECTO. Esos datos podrían diferir en cada ocurrencia de un registro de empleado duplicado porque también dependerían del registro de proyecto padre. Un ejemplo sería un campo con el número de horas por semana que un empleado trabaja en el proyecto. Sin embargo, la mayor parte de los valores de campos en los registros de empleados, como el nombre del empleado, su número de seguro social y su salario, sin duda se duplicarían en cada proyecto al cual perteneciera el empleado. ■

A fin de evitar tal duplicación, contamos con un técnica con la cual podemos especificar varios esquemas jerárquicos en el mismo esquema de base de datos jerárquica. Así podemos definir los vínculos como el tipo de VPH anterior *entre* esquemas jerárquicos, lo que permite sortear el problema de duplicación antes mencionado. Esta técnica, denominada de vínculos "virtuales", se aparta del modelo jerárquico "estricto"; la estudiaremos en la sección 11.2.

11.1.3 *Árboles de ocurrencias jerárquicos*

En correspondencia con un esquema jerárquico, hay muchas ocurrencias jerárquicas en la base de datos. Cada **ocurrencia jerárquica**, llamada también **árbol de ocurrencia**, es una estructura de **árbol** cuya raíz es un solo registro del tipo de registros raíz. Igualmente, el árbol de ocurrencia contiene todas las ocurrencias de registros hijo del registro raíz, todas las ocurrencias de registros hijo dentro de los VPH de cada uno de los registros hijo del registro raíz, y así sucesivamente, hasta los registros de los tipos de registros hoja.

Por ejemplo, consideremos el diagrama jerárquico de la figura 11.5, que representa parte de la base de datos COMPAÑÍA que presentamos en el capítulo 3 y que también usamos en los capítulos 6 a 10. La figura 11.6 muestra un árbol de ocurrencia jerárquico de este esquema. En el árbol de ocurrencia, cada nodo es una ocurrencia de registro, y cada arco representa un vínculo padre-hijo entre dos registros. En las figuras 11.5 y 11.6 utilizamos los caracteres **D, E, P, N, S** y **T** para representar **indicadores de tipo** de los tipos de registros DEPARTAMENTO,

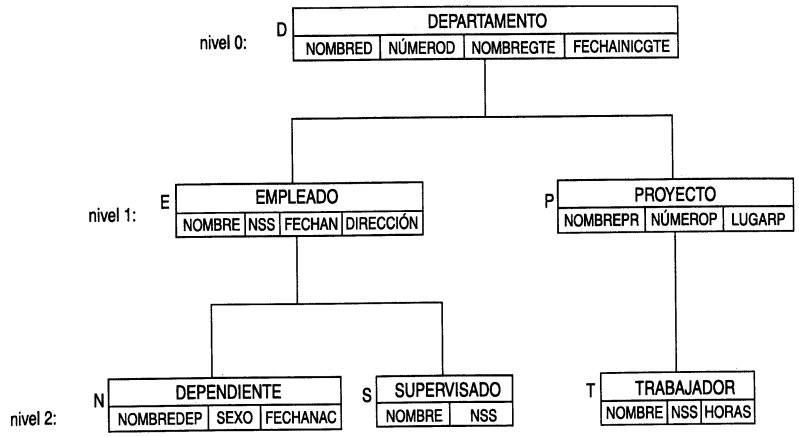


Figura 11.5 Esquema jerárquico de una parte de la base de datos COMPAÑÍA.

EMPLEADO, PROYECTO, DEPENDIENTE, SUPERVISADO y TRABAJADOR, respectivamente. Veremos cuán importantes son estos indicadores de tipo cuando estudiemos las secuencias jerárquicas en la siguiente sección.

Podemos definir los árboles de ocurrencia de manera más formal usando la terminología de las estructuras de árbol, que necesitaremos en la exposición subsecuente. En una estructura de árbol, se dice que la raíz tiene **nivel** cero. El nivel de un nodo no raíz es el nivel de su nodo padre más uno, como se aprecia en las figuras 11.5 y 11.6. Un nodo **descendiente** D de un nodo N es un nodo conectado a N a través de uno o más arcos, de tal modo que el nivel de D es mayor que el de N. Un nodo N y todos sus nodos descendientes forman un **subárbol** del nodo N. Ahora podemos definir un **árbol de ocurrencia** como el subárbol de un registro cuyo tipo es del tipo de registros raíz.

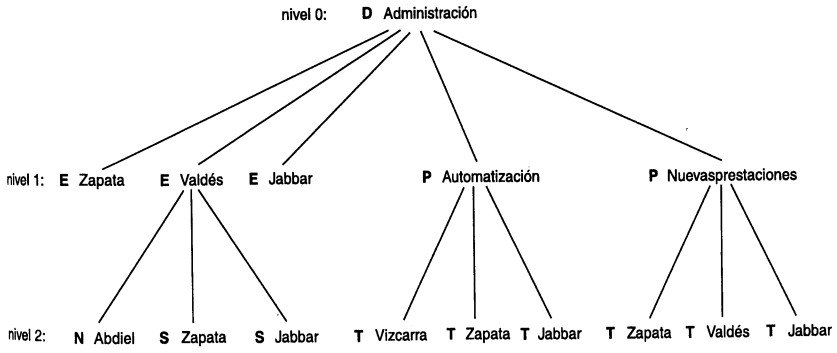


Figura 11.6 Ocurrencia jerárquica (árbol de ocurrencia) del esquema jerárquico de la figura 11.5.

La raíz de un árbol de ocurrencia es una sola ocurrencia de registro del tipo de registros raíz. Puede haber un número variable de ocurrencias de cada tipo de registros no raíz, y cada una de ellas debe tener un registro padre en el árbol de ocurrencia; esto es, cada una de esas ocurrencias debe participar en una ocurrencia de VPH. Observe que cada nodo no raíz, junto con sus nodos descendientes, forma un **subárbol** que por sí solo satisface la estructura de un árbol de ocurrencia para una porción del diagrama jerárquico. Adviértase también que el nivel de un registro en un árbol de ocurrencia es igual al nivel de su tipo de registros en el diagrama jerárquico.

11.1.4 *Forma linealizada de una ocurrencia jerárquica*

Los árboles de ocurrencia jerárquicos se pueden representar en el almacenamiento empleando una de varias estructuras de datos. Sin embargo, una de las estructuras de almacenamiento más simples que podemos usar es el **registro jerárquico**: un ordenamiento lineal de los registros en un árbol de ocurrencia en el *recorrido en preorden* del árbol. Este orden produce una secuencia de ocurrencias de registros denominada **secuencia jerárquica** (o **secuencia jerárquica de registros**) del árbol de ocurrencia; se puede obtener aplicando el siguiente procedimiento recursivo a la raíz de un árbol de ocurrencia:

```
procedimiento Recorrer_en_preorden (registro raíz);
    comenzar
    salida ( registro raíz );
    para cada registro hijo de registro raíz en orden de izquierda a derecha hacer
        Recorrer_en_preorden ( registro hijo )
    fin;
```

El procedimiento anterior, aplicado al árbol de ocurrencia de la figura 11.6, produce la secuencia jerárquica mostrada en la figura 11.7. Si usamos la secuencia jerárquica para implementar los árboles de ocurrencia, necesitaremos almacenar un indicador de tipo de registros con cada registro debido a los diferentes tipos de registros y al número variable de registros hijo en cada vínculo padre_hijo. El sistema necesita examinar el tipo de

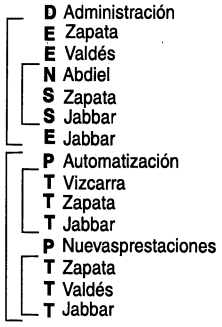


Figura 11.7 Secuencia jerárquica del árbol de ocurrencia de la figura 11.6.

cada registro conforme los recorre secuencialmente. Observe que estos indicadores de tipo de registros son estructuras de implementación que el usuario del SGBD jerárquico no ve.

A menudo la secuencia jerárquica es deseable porque los nodos hijo siguen a su nodo padre en el almacenamiento. Así pues, dado un registro padre, todos los registros descendientes en su subárbol lo siguen en la secuencia jerárquica y se pueden obtener de manera eficiente. Sin embargo, los registros hijo sólo se colocan colectivamente debajo de su registro padre si los registros hijo son nodos hoja en el árbol de ocurrencia; en caso contrario, los subárboles completos de cada nodo hijo se colocan después de su registro padre en orden de izquierda a derecha.

La secuencia jerárquica también es importante porque algunos lenguajes de manipulación de datos jerárquicos, como el que se usa en IMS, la usan como base para definir operaciones de bases de datos jerárquicas. El lenguaje HDML, que veremos en la sección 11.4, se basa en la secuencia jerárquica.

A continuación, definiremos dos términos adicionales empleados por algunos lenguajes jerárquicos. Un **camino jerárquico** es una secuencia de nodos $N_1, N_2, ..., N_i$, donde N_1 es la raíz de un árbol y N_j es un hijo de N_{j-1} para $j = 2, 3, ..., i$. Los caminos jerárquicos se pueden definir sobre un esquema jerárquico o bien sobre un árbol de ocurrencia. Un camino jerárquico es **completo** si N_i es una hoja del árbol. Una **retama** es un conjunto de caminos jerárquicos que resultan del camino jerárquico $N_1, N_2, ..., N_j$, junto con todos los caminos jerárquicos en el subárbol de N_i. Por ejemplo, (DEPARTAMENTO, EMPLEADO, SUPERVISADO) es un camino completo del esquema jerárquico de la figura 11.5. En el árbol de ocurrencia de la figura 11.6, (Administración, Valdés) es un camino, y (Administración, Valdés, {Abdiel, Zapata, Jabbar}) es una retama.

Ahora podemos definir una **ocurrencia de base de datos jerárquica** como una secuencia de todos los árboles de ocurrencia que son ocurrencias de un esquema jerárquico. Esto es similar a la definición de un **bosque** de árboles en las estructuras de datos. Por ejemplo, una ocurrencia de base de datos jerárquica del esquema jerárquico de la figura 11.5 consistiría en varios árboles de ocurrencia similares al de la figura 11.6. Habría un árbol de ocurrencia por cada registro DEPARTAMENTO, y estarían ordenados como el primero, el segundo, ..., el último árbol de ocurrencia.

11.2 Vínculos virtuales padre-hijo

El modelo jerárquico tiene problemas cuando se modelan ciertos tipos de vínculos. Entre ellos están los siguientes vínculos y situaciones:

1. Vínculos M:N.
2. El caso en que un tipo de registros participa como hijo en más de un tipo de VPH.
3. Vínculos n-arios con más de dos tipos de registros participantes.

Como vimos en la sección 11.1.2, el caso 1 puede representarse como un tipo de VPH a expensas de duplicar ocurrencias de registros del tipo de registros hijo. El caso 2 puede representarse en forma similar, con más duplicación de registros. El caso 3 es problemático porque el VPH es un vínculo binario.

La duplicación de registros, además de desperdiciar espacio de almacenamiento, dificulta el mantenimiento de copias consistentes del mismo registro. En el sistema IMS se usa el concepto de tipo de registros virtual (o apuntador) para resolver los tres casos problemáticos

antes identificados. La idea es incluir más de un esquema jerárquico en el esquema de la base de datos jerárquica y usar apuntadores de los nodos de un esquema jerárquico al otro para representar los vínculos. *En vez de seguir* la terminología de IMS, desarrollaremos los conceptos con una perspectiva más general.

Un **tipo de registros virtual** (o **apuntador**) HV es un tipo de registros con la propiedad de que cada uno de sus registros contiene un apuntador a un registro de otro tipo de registros PV. HV siempre desempeña el papel de "hijo virtual" y PV el de "padre virtual" en un "vínculo virtual padre-hijo". Cada ocurrencia de registro *h* de HV apunta a una y sólo una ocurrencia de registro *p* de PV. En vez de duplicar el registro *p* mismo en un árbol de ocurrencia, incluimos el registro virtual *h* que contiene un apuntador a *p*. Varios registros virtuales pueden apuntar a *p*, pero sólo se almacenará una copia de *p* en la base de datos.

La figura 11.8 muestra el vínculo M:*N* entre EMPLEADO y PROYECTO representado con registros virtuales APUNTE y APUNTP. Compare esto con la figura 11.4, donde el mismo vínculo se representó sin registros virtuales. La figura 11.9 muestra los árboles de ocurrencia y los apuntadores para los ejemplares de datos mostrados en el ejemplo 1 cuando se usa el esquema jerárquico de la figura 11.8(a). En la figura 11.9 sólo hay una copia de cada registro EMPLEADO; sin embargo, varios registros virtuales pueden apuntar al mismo registro EMPLEADO. Así, la información almacenada en dicho registro no se duplicará. La información que dependa tanto de los registros padre como hijo —como las horas por semana que un trabajador dedica a un proyecto— se incluirá en el registro apuntador virtual; los usuarios de bases de datos jerárquicas suelen llamar **datos de intersección** a tales datos.

Observe que el vínculo entre EMPLEADO y APUNTE de la figura 11.8(a) es un vínculo 1:*N* y por tanto califica como tipo de VPH. Este tipo de vínculos se denomina **tipo de vínculos padre-hijo virtual** (VPHV). EMPLEADO es el **padre virtual** de APUNTE, y éste es el **hijo virtual** de EMPLEADO. En términos conceptuales, los tipos de VPH y los tipos de VPHV son similares; la principal diferencia entre ellos radica en la forma en que se implementan. Los tipos de VPH casi siempre se implementan por medio de la secuencia jerárquica, en tanto que los tipos de VPHV suelen implementarse estableciendo un apuntador (uno físico que contenga una dirección o uno lógico que contenga una clave) desde un registro hijo virtual a su registro padre virtual. Esto afecta principalmente la eficiencia de ciertas consultas.

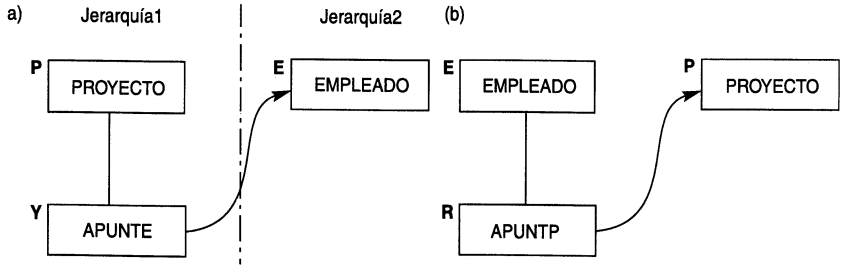


Figura 11.8 Representación de un vínculo M:*N* con tipos de VPHV. (a) Una representación del vínculo M:*N*, con padre virtual EMPLEADO. (b) Representación alternativa del vínculo M:*N*, con padre virtual PROYECTO.

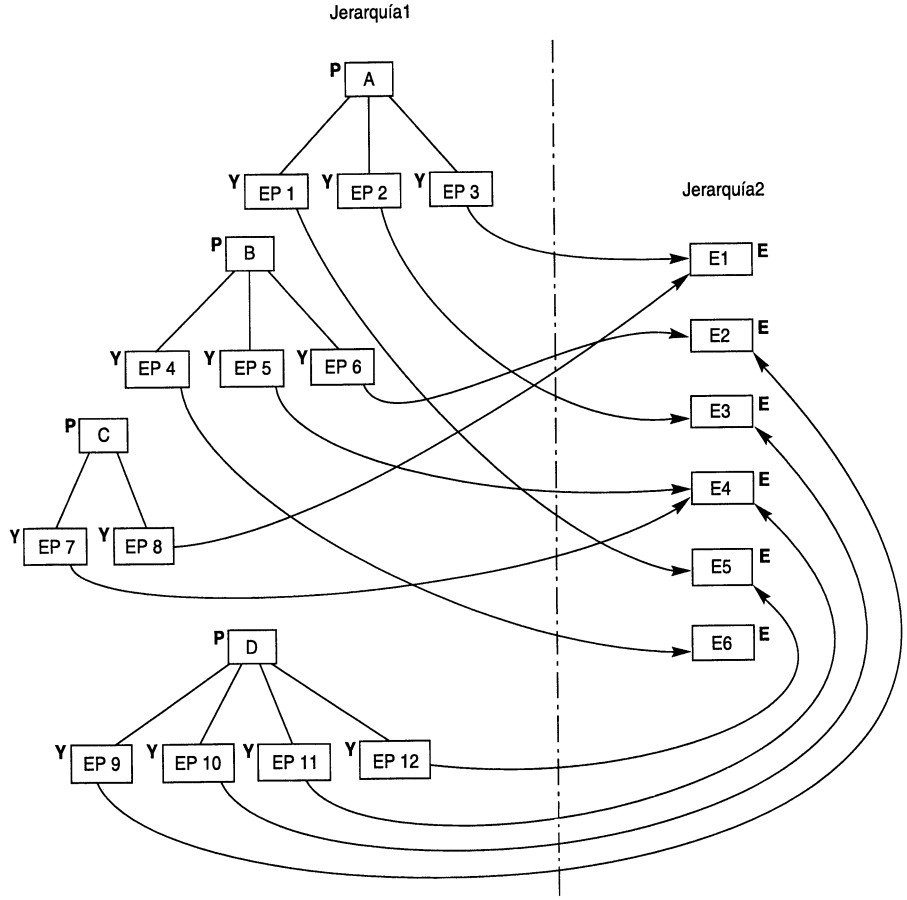


Figura 11.9 Las ocurrencias del ejemplo 1 correspondientes al esquema jerárquico de la figura 11.8(a).

La figura 11.10 muestra un esquema de base de datos jerárquica para la base de datos COMPAÑÍA usando algunos VPHV y sin redundancia en sus ocurrencias de registros. El esquema de base de datos se compone de dos esquemas jerárquicos: uno con DEPARTAMENTO como raíz y el otro con EMPLEADO como raíz. Se incluyen cuatro VPHV, todos con EMPLEADO como padre virtual, para representar los vínculos sin redundancia. Cabe señalar que IMS *tal vez no permita esto* porque una restricción de implementación en IMS hace que un registro sólo pueda ser padre virtual de cuando más un VPHV; a fin de sortear esta restricción, podemos crear tipos de registros hijo de EMPLEADO ficticios en la jerarquía 2 de modo que cada VPHV apunte a un tipo de registros padre virtual distinto.

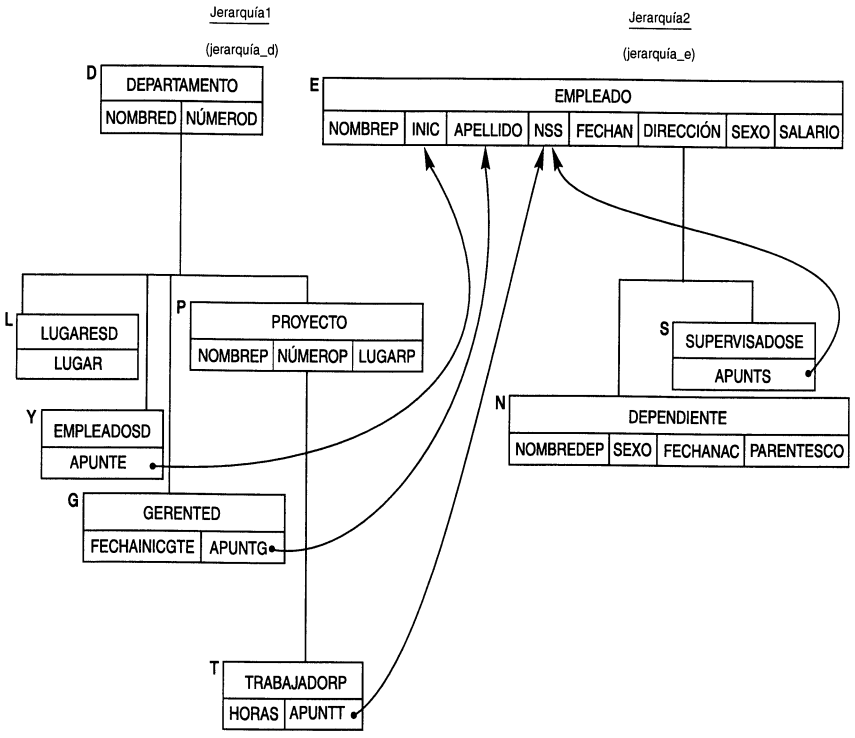


Figura 11.10 Esquema jerárquico para la base de datos COMPAÑÍA, usando tipos de VPHV entre dos jerarquías para eliminar los ejemplares de registros redundantes.

En general, hay muchos métodos factibles para diseñar una base de datos empleando el modelo jerárquico. En muchos casos, las consideraciones de rendimiento son el factor más importante para elegir un esquema de base de datos jerárquica en vez de otro. El rendimiento depende de las opciones de implementación disponibles en cada sistema, así como de límites específicos que el DBA establece en una instalación en particular: por ejemplo, si el sistema provee o no ciertos tipos de apuntadores y si el DBA impone ciertos límites sobre el número de niveles.

Algo que debemos tener presente respecto a los VPHV es que se pueden implementar de diversas maneras. Una opción consiste simplemente en tener un apuntador al padre virtual en cada hijo virtual, como vimos antes. Una segunda opción es tener, además del apuntador hijo-a-padre, un enlace hacia atrás del padre virtual a una lista enlazada de registros hijo virtuales. El apuntador del padre virtual al primer registro hijo virtual se denomina **apuntador a hijo virtual**, y un apuntador de un hijo virtual al siguiente recibe el nombre de **apuntador a gemelo virtual**. En este caso, el modelo jerárquico se vuelve *muy parecido* al modelo de red que vimos en el capítulo 10. Este enlace hacia atrás facilita la obtención de todos los registros hijo virtuales a partir de un registro padre virtual dado.

11.3 Restricciones de integridad en el modelo jerárquico

Siempre que especificamos un esquema jerárquico, habrá varias **restricciones inherentes** al modelo jerárquico. Entre ellas se cuentan:

1. Ninguna ocurrencia de registro, con excepción de los registros raíz, puede existir si no está relacionada con una ocurrencia de registro padre. Esto tiene las siguientes implicaciones:
   a. Ningún registro hijo puede insertarse si no está enlazado a un registro padre.
   b. Un registro hijo se puede eliminar independientemente de su padre; pero la eliminación de un padre causa automáticamente la eliminación de todos sus registros hijos y descendientes.
   c. Las reglas anteriores no se aplican a los registros hijo y padre virtuales. La regla en este caso es que un apuntador en un registro hijo virtual debe apuntar a una ocurrencia real de un registro padre virtual. No debe permitirse la eliminación de un registro en tanto existan apuntadores a él en registros hijo virtuales, lo que lo convierte en un registro padre virtual.
2. Si un registro hijo tiene dos o más registros padre del *mismo* tipo de registros, el registro hijo debe duplicarse una vez bajo cada registro padre.
3. Un registro hijo que tenga dos o más registros padre de *diferentes* tipos de registros sólo puede tener un padre real; todos los demás deben representarse como padres virtuales.

Por añadidura, cada SGBD puede tener sus propias reglas de integridad adicionales únicas para su implementación. En IMS, por ejemplo, un tipo de registros puede ser el padre virtual en *sólo un* tipo de VPHV. Esto implica que el esquema de la figura 11.10 *no* está permitido en IMS, porque el registro EMPLEADO es padre virtual en cuatro VPHV distintos. Otra regla en IMS es que un tipo de registros raíz *no puede* ser un tipo de registros hijo virtual en un tipo de VPHV.

Cualesquier otras restricciones que no estén implícitas en un esquema jerárquico las deberán imponer explícitamente los programadores en los programas de actualización de la base de datos. Por ejemplo, si se actualiza un registro duplicado, es el programa de actualización el que debe asegurarse de que todas las copias se actualicen de la misma manera.

11.4 Definición de datos en el modelo jerárquico*

En esta sección presentaremos un ejemplo de lenguaje de definición de datos jerárquico (HDDL: *hierarchical data definition language*), que *no* es el lenguaje de ningún SGBD jerárquico específico, pero que nos permitirá ilustrar los conceptos lingüísticos de una base de datos jerárquica. El HDDL ilustra cómo puede definirse un esquema de base de datos jerárquica. Parte de la terminología que se usa aquí es diferente de la de IMS y de la de otros SGBD jerárquicos. Para definir un esquema de base de datos jerárquica, debemos definir los campos de cada tipo de registros, el tipo de datos de cada campo y cualesquier restricciones de clave sobre los campos. Además, debemos especificar un tipo de registros raíz como tal; y para cada tipo de registros no raíz, será preciso especificar su padre (real) en un tipo de VPH. También tendremos que especificar todos los VPHV.

La figura 11.11 muestra la especificación en HDDL del esquema de base de datos de la figura 11.10. La mayor parte de los enunciados no requieren mayor explicación. En los SGBD jerárquicos reales, la sintaxis suele ser más compleja y, como mencionamos antes, la terminología puede ser diferente. Observe también que algunas de las estructuras, como la representación de EMPLEADO como padre virtual en más de un tipo de VPHV, podría estar prohibida en algunos SGBD jerárquicos, como IMS.

En la figura 11.11, cada tipo de registros se declara como tipo raíz o bien se declara un solo tipo de registros padre (real) para ese tipo de registros. En seguida se listan los elementos de información del registro junto con sus tipos de datos. Debemos especificar un padre virtual para los elementos de información de tipo *apuntador* (*pointer*). Los elementos de información declarados por la cláusula KEY (clave) deben tener valores únicos para cada registro. Cada cláusula KEY especifica una clave independiente; si una de estas cláusulas lista más de un campo, la combinación de los valores de estos campos debe ser única en cada registro.

La cláusula CHILD NUMBER (número de hijo) especifica el orden de izquierda a derecha de un tipo de registros hijo bajo su tipo de registros padre (real). En la figura 11.11 esto corresponde al orden de izquierda a derecha que se muestra en la figura 11.10 y es necesaria para especificar el orden de los subárboles hijos de diferentes tipos de registros hijo bajo un registro padre en la secuencia jerárquica. Por ejemplo, debajo de un registro EMPLEADO tenemos primero todos los subárboles de sus registros hijo DEPENDIENTE (CHILD NUMBER = 1), y luego todos los subárboles de sus registros hijo SUPERVISADOSE (CHILD NUMBER = 2) en la secuencia jerárquica.

La cláusula ORDER BY (ordenar por) especifica el orden de los registros individuales del mismo tipo en la secuencia jerárquica. En el caso de un tipo de registros raíz, esto especifica el orden de los árboles de ocurrencia. Por ejemplo, los registros EMPLEADO están ordenados alfabéticamente según estos campos. En el caso de tipos de registros no raíz, la cláusula ORDER BY indica cómo deben ordenarse los registros *dentro de cada registro padre*, especificando un campo llamado **clave de secuencia**. Por ejemplo, los registros PROYECTO controlados por un cierto DEPARTAMENTO tendrán sus subárboles en orden alfabético dentro del mismo registro DEPARTAMENTO padre por NOMBREPR, de acuerdo con la figura 11.11.

11.5 Uso de la transformación ER-jerárquico para el diseño de bases de datos jerárquicas*

En el modelo jerárquico, sólo los tipos de vínculos 1:*N* pueden representarse en una jerarquía específica como tipos de vínculos padre-hijo (VPH). Por añadidura, un tipo de registros puede tener como máximo un tipo de registros padre (real); por tanto, los tipos de vínculos M:*N* son difíciles de representar. Entre las posibles formas de representar los tipos de vínculos M:*N* en una base de datos jerárquica están las siguientes:

- Representar el tipo de vínculos M:*N* como si fuera 1:*N*. En este caso, los ejemplares de registros del lado *N* se duplicarán porque cada uno puede estar relacionado con varios padres. Esta representación mantiene todos los tipos de registros en una sola jerarquía *a expensas de duplicar ejemplares de registros*. Los programas de aplicación que actualizan la base de datos deben mantener la consistencia de las copias.


```
SCHEMA NAME = COMPAÑÍA

HIERARCHIES = JERARQUÍA1, JERARQUÍA2

RECORD
    NAME = EMPLEADO
    TYPE = ROOT OF JERARQUÍA2
    DATA ITEMS =
        NOMBREP     CHARACTER 15
        INIC        CHARACTER 1
        APELLIDO    CHARACTER 15
        NSS         CHARACTER 9
        FECHAN      CHARACTER 9
        DIRECCIÓN   CHARACTER 30
        SEXO        CHARACTER 1
        SALARIO     CHARACTER 10
    KEY = NSS
    ORDER BY APELLIDO, NOMBREP

RECORD
    NAME = DEPARTAMENTO
    TYPE = ROOT OF JERARQUÍA1
    DATA ITEMS =
        NOMBRED     CHARACTER 15
        NÚMEROD     INTEGER
    KEY = NOMBRED
    KEY = NÚMEROD
    ORDER BY NOMBRED

RECORD
    NAME = LUGARESD
    PARENT = DEPARTAMENTO
    CHILD NUMBER = 1
    DATA ITEMS =
        LUGAR       CHARACTER 15

RECORD
    NAME = GERENTED
    PARENT = DEPARTAMENTO
    CHILD NUMBER = 3
    DATA ITEMS =
        FECHAINICGTE    CHARACTER 9
        APUNTG          POINTER WITH VIRTUAL PARENT = EMPLEADO

RECORD
    NAME = PROYECTO
    PARENT = DEPARTAMENTO
    CHILD NUMBER = 4
    DATA ITEMS =
        NOMBREPR    CHARACTER 15
        NÚMEROP     INTEGER
        LUGARP      CHARACTER 15
    KEY = NOMBREPR
    KEY = NÚMEROP
    ORDER BY NOMBREPR
```

Figura 11.11 Declaraciones para el esquema jerárquico de la figura 11.10.
(*continúa en la siguiente página*)

```
RECORD
    NAME = TRABAJADORP
    PARENT = PROYECTO
    CHILD NUMBER = 1
    DATA ITEMS =
        HORAS       CHARACTER 4
        APUNTT      POINTER WITH VIRTUAL PARENT = EMPLEADO

RECORD
    NAME = EMPLEADOSD
    PARENT = DEPARTAMENTO
    CHILD NUMBER = 2
    DATA ITEMS =
        APUNTE      POINTER WITH VIRTUAL PARENT = EMPLEADO

RECORD
    NAME = SUPERVISADOSE
    PARENT = EMPLEADO
    CHILD NUMBER = 2
    DATA ITEMS =
        APUNTS      POINTER WITH VIRTUAL PARENT = EMPLEADO

RECORD
    NAME = DEPENDIENTE
    PARENT = EMPLEADO
    CHILD NUMBER = 1
    DATA ITEMS =
        NOMBREDEP   CHARACTER 15
        SEXO        CHARACTER 1
        FECHANAC    CHARACTER 9
        PARENTESCO  CHARACTER 10
        ORDER BY DESC FECHANAC
```

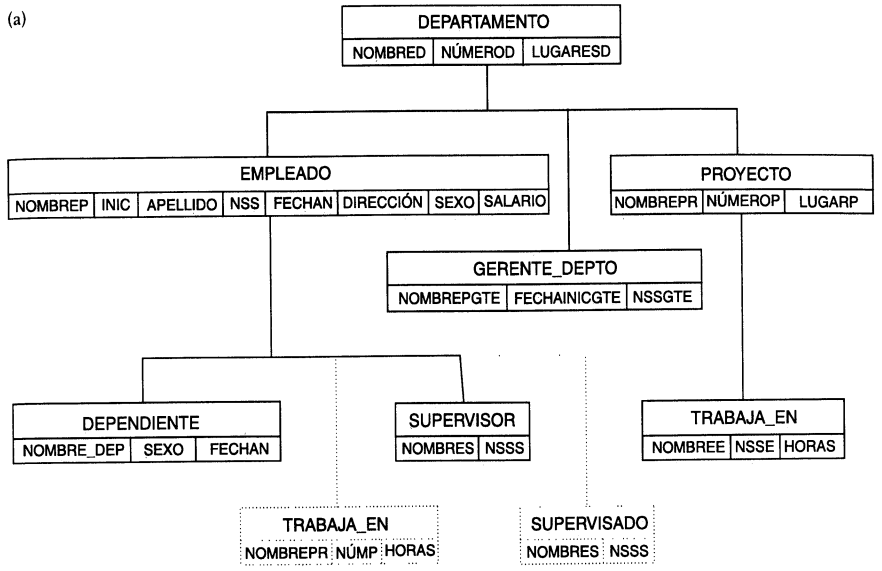
Figura 11.11 (*continuación*) Declaraciones para el esquema jerárquico de la figura 11.10.

- Crear más de una jerarquía y tener tipos de vínculos padre-hijo virtuales (VPHV) (apuntadores lógicos) a partir de un tipo de registros que aparece en una jerarquía al tipo de registros raíz de otra jerarquía. Estos apuntadores pueden servir para representar el tipo de vínculos M:N en forma similar a la empleada en el modelo de red. Aun así, una restricción, adoptada del modelo del SGBD en IMS, impide que un tipo de registros tenga más de un hijo virtual.

Como se deben considerar múltiples opciones, no existe un método estándar para transformar un esquema ER a un esquema jerárquico. Ilustraremos las dos posibilidades antes analizadas con dos esquemas jerárquicos (véanse las Figs. 11.12(a) y (b)), con los que podemos representar el esquema ER de la figura 3.2. Una tercera alternativa es la que se muestra en la figura 11.10.

La figura 11.12(a) muestra una sola jerarquía que puede servir para representar el esquema ER de la figura 3.2. Escogemos DEPARTAMENTO como tipo de registros raíz de la jerarquía. Los tipos de vínculos 1:*N* PERTENECE_A y CONTROLA, así como el tipo de vínculos 1:1 DIRIGE, se representan en el primer nivel de la jerarquía mediante los tipos de registros EMPLEADO, PROYECTO y GERENTE_DEPTO, respectivamente. No obstante, a fin de limitar la redundancia, sólo mantendremos algunos de los atributos de un empleado que es gerente en

(a)



(b)

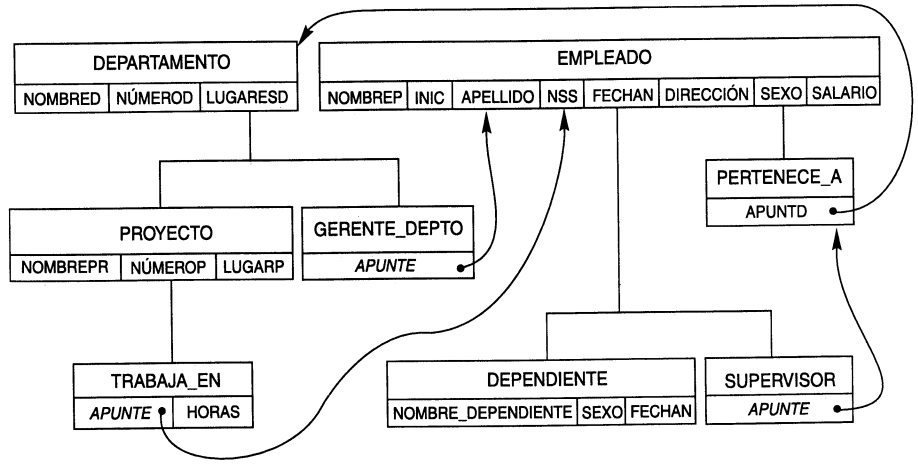


Figura 11.12 Transformación del esquema ER de la figura 3.2 al modelo jerárquico. (a) Esquema jerárquico de la base de datos COMPAÑÍA con una sola jerarquía. (b) Otro esquema jerárquico para la misma base de datos con dos jerarquías y cuatro VPHV.

el tipo de registros GERENTE_DEPTO. Los registros EMPLEADO de un árbol jerárquico propiedad de un registro DEPARTAMENTO dado representarán a los empleados que pertenecen a ese departamento. De manera similar, los registros PROYECTO representarán los proyectos controlados por ese departamento, y el registro GERENTE_DEPTO representará al empleado que dirige dicho departamento. Así, un empleado que es gerente se representa dos veces: una como ejemplar de EMPLEADO y la segunda como ejemplar de GERENTE_DEPTO. Los programas de aplicación tienen la obligación de mantener estas copias en un estado consistente.

El tipo de vínculos 1:*N* DEPENDIENTES_DE se representa con el tipo de registros DEPENDIENTE como subordinado de EMPLEADO (aquí no se incluyó el atributo PARENTESCO de DEPENDIENTE). El tipo de vínculos *M*:*N* TRABAJA_EN se representa como subordinado de PROYECTO, pero sólo se incluyen el NOMBREE y el NSSE del empleado en TRABAJA_EN, junto con el atributo HORAS. Un empleado que trabaje en un número determinado de proyectos distintos se almacenará en ese mismo número de copias de ejemplares de registro TRABAJA_EN con valores idénticos de NOMBREE y NSSE. El resto de la información sobre el empleado no se repetirá en TRABAJA_EN, a fin de abatir la redundancia. Observe que cada registro TRABAJA_EN representa uno de los empleados que trabajan en un proyecto dado. Como alternativa, podríamos representar TRABAJA_EN como un tipo de registros subordinado de EMPLEADO, en cuyo caso cada registro TRABAJA_EN representaría uno de los proyectos en los que trabaja un empleado, y sus campos serían NOMBREPR, NÚMEROP y HORAS, como se indica en la figura 11.12(a) con el cuadro de líneas punteadas de TRABAJA_EN. En el segundo caso, la información de PROYECTO se duplicaría en varias copias de los registros TRABAJA_EN.

Por último, el tipo de vínculos SUPERVISIÓN se representa como subordinado de EMPLEADO. Podríamos optar por representarlo en el papel de supervisor o bien en el de supervisado. En la figura 11.12(a) cada registro SUPERVISOR representa al supervisor del registro EMPLEADO propietario en la jerarquía, así que el vínculo jerárquico representa el papel de supervisor; cada empleado tiene un solo registro SUPERVISOR hijo que representa a su supervisor directo. Como alternativa, podríamos representar el papel de supervisado en el vínculo jerárquico; en tal caso, cada registro EMPLEADO que representara a un empleado supervisor se relacionaría con (posiblemente) muchos supervisados directos como registros hijo. Esto se muestra con líneas punteadas en el cuadro SUPERVISADO de la figura 11.12(a). En ambos casos, la información de EMPLEADO se repite en los registros SUPERVISOR o SUPERVISADO, por lo que sólo incluimos unos cuantos de los atributos de EMPLEADO.

En la figura 11.12(a) hay una repetición excesiva de información sobre los empleados en los tipos de registros EMPLEADO, GERENTE_DEPTO, TRABAJA_EN y SUPERVISOR, porque todos representan a empleados en diversos papeles. Podemos limitar un tanto la repetición si representamos el esquema de la figura 3.2 con dos o más jerarquías. En la figura 11.12(b) intervienen dos jerarquías. La primera tiene DEPARTAMENTO como tipo de registros raíz y representa los tipos de vínculos CONTROLA, DIRIGE y TRABAJA_EN. La segunda jerarquía tiene EMPLEADO como tipo de registros raíz y representa los tipos de vínculos DEPENDIENTES_DE y SUPERVISIÓN. Si usamos vínculos padre-hijo virtuales y apuntadores virtuales en los tipo de registros TRABAJA_EN, GERENTE_DEPTO y SUPERVISOR, no repetiremos la información de ningún empleado. Cada apuntador apuntará a un registro EMPLEADO, pero la información sobre el empleado se almacenará una sola vez como registro raíz de la segunda jerarquía. El tipo de vínculos PERTENECE_A se representa con un hijo del registro EMPLEADO, llamado PERTENECE_A, cuyo padre virtual es DEPARTAMENTO.†

†Esto se hace porque en IMS el registro raíz EMPLEADO *no puede* tener un padre virtual.

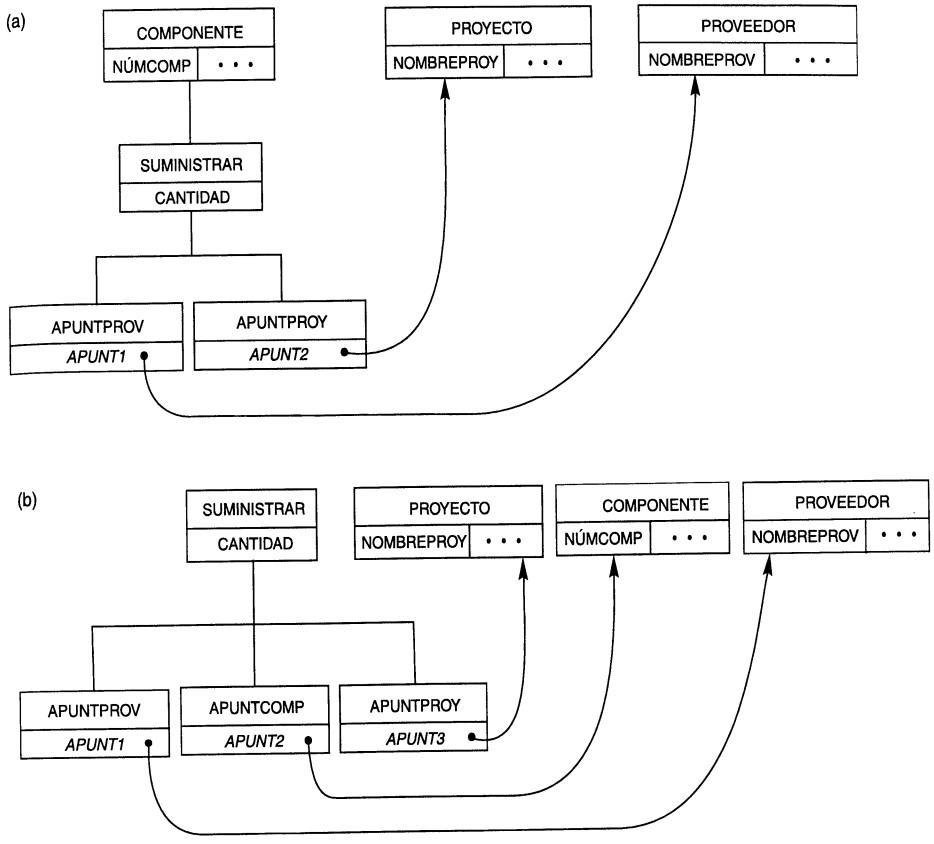


Figura 11.13 Transformación de un tipo de vínculos *n*-ario ($n = 3$) SUMINISTRAR del modelo ER al jerárquico. (a) Una opción para representar un vínculo ternario. (b) Representación de un vínculo ternario con tres tipos de VPHV.

Por último, consideremos la transformación de los tipos de vínculos *n*-arios, con $n > 2$. La figura 11.13 muestra dos opciones para establecer la transformación del tipo de vínculos ternario SUMINISTRAR de la figura 3.16(a). Debido a la restricción, derivada de IMS, por la cual un tipo de registros no puede tener más de un padre virtual, no podemos colocar SUMINISTRAR debajo de COMPONENTE, digamos, e incluir dos apuntadores a dos padres virtuales PROYECTO y PROVEEDOR. La opción de la figura 11.13(a) crea *dos tipos de registros apuntadores* bajo SUMINISTRAR con los padres virtuales PROYECTO y PROVEEDOR. Otra opción, como se ve en la figura 11.13(b), consistiría en que SUMINISTRAR fuera raíz de una jerarquía y se crearan tres tipos de registros apuntadores bajo ella que apuntaran a los tipos de registros participantes como padres virtuales. Esta opción es la que ofrece más flexibilidad.

Es evidente que el modelo jerárquico ofrece muchas opciones para representar el mismo esquema ER, y podríamos haber empleado muchas otras representaciones además de las que vimos. Las cuestiones de acceso eficiente a los datos y de limitar la redundancia vs. facilitar la obtención son importantes al elegir una representación específica. En general, se considera que el modelo jerárquico es inferior, en cuanto a su capacidad de modelado, que el modelo relacional y el modelo de red, por las siguientes razones:

- Los tipos de vínculos M:*N* sólo pueden representarse añadiendo registros redundantes o usando vínculos padre-hijo virtuales y registros apuntadores.
- Todos los tipos de vínculos 1:*N* en una jerarquía se deben mantener en la misma dirección.
- Un tipo de registros en una jerarquía puede tener cuando más un tipo de registros propietario (real).
- Un tipo de registros puede tener cuando más dos padres: uno real y uno virtual. (Esta limitación es específica de IMS.)

11.6 Lenguaje de manipulación de datos para el modelo jerárquico*

A continuación estudiaremos el HDML (*Hierarchical Data Manipulation Language*: lenguaje de manipulación de datos jerárquicos), que es un lenguaje de registro por registro para manipular bases de datos jerárquicas. Las órdenes del lenguaje deben estar incorporadas en un lenguaje de programación de aplicación general, el denominado lenguaje **anfitrión**. Aunque lo más común es que COBOL o PL/1 sean los lenguajes anfitriones, usaremos PASCAL en nuestros ejemplos para mantener la congruencia con el resto del libro. Cabe señalar que HDML *no* es un lenguaje para un SGBD jerárquico en particular; con él, aquí sólo queremos ilustrar los conceptos de un lenguaje de manipulación de bases de datos jerárquicas. Para empezar, presentaremos el concepto general de área de trabajo del usuario para su comunicación con el sistema, junto con algunos conceptos de indicadores de actualidad.

11.6.1 *Área de trabajo del usuario (UWA) y conceptos de indicadores de actualidad para usar órdenes de HDML*

En un lenguaje de registro por registro, una operación de obtención de base de datos coloca los registros obtenidos en **variables de programa**. En nuestros ejemplos, los registros de base de datos obtenidos se colocarán en variables de programa PASCAL. Así, el programa puede hacer referencia a estas variables para tener acceso a los valores de los campos de los registros. Suponemos que se ha declarado un tipo de registros PASCAL para cada uno de los tipos de registros que aparecen en el esquema de la figura 11.10. Las variables de programa PASCAL, que se muestran en la figura 11.14, utilizan los *mismos nombres de campos* que los del esquema de base de datos de la figura 11.10, pero los *nombres de registros* llevan el prefijo P_. Estas variables de programa existen en lo que a menudo recibe el nombre de **área de trabajo del usuario**. Observe que es posible declarar automáticamente estas variables haciendo referencia al esquema de base de datos declarado en la figura 11.11. En un principio, los valores de estas variables de registro no estarán definidos. Siempre que una operación de obtención de datos lea un registro de un cierto tipo, lo colocará en la variable de programa UWA correspondiente.


```
var P_EMPLEADO      : record
                      NOMBREP: packed array [1..15] of char;
                      INIC: char;
                      APELLIDO: packed array [1..15] of char;
                      NSS: packed array [1..9] of char;
                      FECHAN: packed array [1..9] of char;
                      DIRECCIÓN: packed array [1..30] of char;
                      SEXO: char;
                      SALARIO: packed array [1..10] of char
                      end;
    P_DEPARTAMENTO: record
                      NOMBRED: packed array [1..15] of char;
                      NÚMEROD: integer
                      end;
    P_LUGARESD    : record
                      LUGAR: packed array [1..15] of char
                      end;
    P_GERENTED    : record
                      FECHAINICGTE: packed array [1..9] of char;
                      APUNTG: database pointer to EMPLEADO
                      end;
    P_PROYECTO    : record
                      NOMBREPR: packed array [1..15] of char;
                      NÚMEROP: integer;
                      LUGARP: packed array [1..15] of char
                      end;
    P_TRABAJADORP : record
                      HORAS: packed array [1..4] of char;
                      APUNTT: database pointer to EMPLEADO
                      end;
    P_EMPLEADOSD  : record
                      APUNTE: database pointer to EMPLEADO
                      end;
    P_SUPERVISADOE: record
                      APUNTS: database pointer to EMPLEADO
                      end;
    P_DEPENDIENTE : record
                      NOMBREDEP: packed array [1..15] of char;
                      SEXO: char;
                      FECHANAC: packed array [1..9] of char;
                      PARENTESCO: packed array [1..10] of char
                      end;
```

Figura 11.14 Variables de programa PASCAL en el UWA correspondientes a parte del esquema jerárquico de la figura 11.10.

El HDML se basa en el concepto de **secuencia jerárquica** definido en la sección 11.1. Después de cada orden de base de datos, el último registro al que tuvo acceso la orden es el **registro actual de la base de datos**. El SGBD mantiene un apuntador al registro actual. Las órdenes de base de datos subsecuentes proceden *del registro actual* y pueden definir un nuevo registro actual, según el tipo de la orden. Así pues, las órdenes de HDML recorren la base de datos jerárquica obteniendo los registros solicitados por la consulta. Originalmente, el registro actual de la base de datos es un "registro imaginario", situado justo antes del registro raíz del primer árbol de ocurrencia en la base de datos.

Si una base de datos contiene más de un esquema jerárquico, y estas jerarquías se procesan juntas, con el sistema IMS se podrá definir una vista de usuario para crear un esquema

jerárquico personalizado que incluya los tipos de registros que se desea conectar mediante tipos de VPHV.† Estas vistas se tratan como un solo **esquema jerárquico** y cada una tiene su propio **registro actual de base de datos.** Puesto que no es nuestra intención detallar la definición y el proceso de tales vistas, supondremos, por comodidad, que *cada esquema jerárquico* tiene su propio **registro actual de jerarquía.** IMS también puede recordar el último registro *de cada tipo de registros* al que tuvo acceso, por lo que supondremos que el sistema sigue la pista del **actual de tipo de registros** para *cada tipo de registros*: éste es un apuntador al último registro de ese tipo al que se tuvo acceso. Las órdenes de HDML se refieren implícitamente a estos tres tipos de **indicadores de actualidad**:

- Actual de base de datos.
- Actual de jerarquía para cada esquema jerárquico.
- Actual de tipo de registros para cada tipo de registros.

La programación registro por registro requiere una interacción continua entre el programa usuario y el SGBD. La **información de estado** que cada orden de base de datos tiene al final se debe comunicar de vuelta al programa, y esto se hace mediante una variable llamada **DB_STATUS,** cuyo valor lo establece el software de SGBD después de ejecutar cada orden de base de datos. Supondremos que un valor de DB_STATUS = 0 especifica que la última orden se ejecutó con éxito.

Las órdenes de HDML se pueden clasificar como órdenes de obtención de datos, órdenes de actualización y órdenes de retención de actualidad. Las **órdenes de obtención de datos** leen uno o más registros de la base de datos colocándolos en las variables correspondientes, y pueden modificar algunos indicadores de actualidad. Las **órdenes de actualización** sirven para insertar, eliminar y modificar registros de la base de datos. Las **órdenes de retención de actualidad** se emplean para marcar el registro actual de modo que una orden subsecuente lo pueda actualizar o eliminar. La tabla 11.1 muestra un resumen de las órdenes de HDML.

A continuación estudiaremos cada una de estas órdenes e ilustraremos nuestro análisis con ejemplos basados en el esquema de la figura 11.10. En los segmentos de programa, *las órdenes de HDML llevan un signo $ como prefijo* para distinguirlas de las instrucciones del lenguaje PASCAL. Las palabras reservadas de este último —como if, then, while y for— se escribirán en minúsculas.

11.6.2 *La orden GET*

La orden de HDML para obtener un registro es **GET.** Esta orden tiene muchas variaciones; la estructura de dos de ellas es la siguiente, donde las partes opcionales se encierran entre corchetes [...]:

- GET FIRST†† <nombre de tipo de registros> [WHERE <condición>]
- GET NEXT <nombre de tipo de registros> [WHERE <condición>]

La variación más simple es la orden **GET FIRST** (obtener el primero), que siempre comienza a buscar en la base de datos a partir del *principio de la secuencia jerárquica* hasta encontrar la primera ocurrencia de registro del <nombre de tipo de registros> que satisface <condición>. Este registro también se convierte en el actual de base de datos,

†En IMS estos esquemas de "vistas" se llaman **bases de datos lógicas** y en la sección 11.7.3 los estudiaremos brevemente.
††Esto es similar a la orden **GET UNIQUE** (GU: obtener único) de IMS.

Tabla 11.1 Resumen de órdenes de HDML

OBTENCIÓN DE DATOS	
GET	OBTENER UN REGISTRO, COLOCARLO EN LA VARIABLE DE PROGRAMA CORRESPONDIENTE Y CONVERTIRLO EN EL REGISTRO ACTUAL. LAS VARIACIONES INCLUYEN GET FIRST, GET NEXT, GET NEXT WITHIN PARENT Y GET PATH.
ACTUALIZACIÓN DE REGISTROS	
INSERT	ALMACENAR UN NUEVO REGISTRO EN LA BASE DE DATOS Y CONVERTIRLO EN EL REGISTRO ACTUAL
DELETE	ELIMINAR DE LA BASE DE DATOS EL REGISTRO ACTUAL (Y SU SUBÁRBOL)
REPLACE	MODIFICAR ALGUNOS CAMPOS DEL REGISTRO ACTUAL
RETENCIÓN DE ACTUALIDAD	
GET HOLD	OBTENER UN REGISTRO Y MANTENERLO COMO REGISTRO ACTUAL PARA PODERLO ELIMINAR O REEMPLAZAR POSTERIORMENTE

actual de jerarquía y actual de tipo de registros, y se coloca en la variable de programa UWA correspondiente. Por ejemplo, si queremos obtener el "primer" registro EMPLEADO de la secuencia jerárquica cuyo nombre sea José Silva, escribiremos EJ1:

```
EJ1: $GET FIRST EMPLEADO WHERE NOMBREP='José' AND APELLIDO='Silva';
```

El SGBD usa la condición que sigue a WHERE para buscar el primer registro, en el orden de la secuencia jerárquica, que satisface la condición y pertenece al tipo de registros especificado. El valor de DB_STATUS se pone en cero *si la búsqueda tiene éxito*; en caso contrario, DB_STATUS adopta algún otro valor —digamos, 1— que indica *no se encontró*. Otros errores o excepciones se indican con otros valores de DB_STATUS.

Si más de un registro de la base de datos satisface la condición WHERE y queremos obtenerlos todos, deberemos escribir una construcción cíclica en el programa anfitrión y usar la orden GET NEXT (obtener el siguiente). Suponemos que esta orden inicia su búsqueda a partir del *registro actual del tipo de registros especificado en* GET NEXT† y busca hacia adelante en la secuencia jerárquica hasta encontrar otro registro del tipo especificado que satisfaga la condición WHERE. Por ejemplo, si queremos obtener los registros de todos los empleados cuyo salario sea menor que $20 000 e imprimir un listado de sus nombres, podemos escribir un segmento de programa como el de EJ2:

```
EJ2: $GET FIRST EMPLEADO WHERE SALARIO < '20000.00';
     while DB_STATUS = 0 do
          begin
          writeln ( P_EMPLEADO.NOMBREP, P_EMPLEADO.APELLIDO );
          $GET NEXT EMPLEADO WHERE SALARIO < '20000.00'
          end;
```

†Por lo regular, las órdenes de IMS parten del *actual de base de datos*, no del actual del tipo de registros especificado, como ocurre con las órdenes de HDML.

En EJ2, el ciclo *while* continúa hasta que no se encuentran más registros EMPLEADO en la base de datos que satisfagan la condición WHERE; por tanto, la búsqueda continúa hasta el último registro de la base de datos (en la secuencia jerárquica). Cuando no se encuentren más registros, DB_STATUS adquirirá un valor distinto de cero (un código que indica "se llegó al final de la base de datos") y el ciclo concluirá. Cabe señalar que la condición WHERE en las órdenes GET es opcional. Si no hay condición, se obtiene el siguiente registro, en el orden de la secuencia jerárquica, del tipo de registros especificado. Por ejemplo, para obtener todos los registros EMPLEADO de la base de datos, podemos usar EJ3:

```
EJ3: $GET FIRST EMPLEADO;
     while DB_STATUS = 0 do
          begin
          writeln ( P_EMPLEADO.NOMBREP, P_EMPLEADO.APELLIDO );
          $GET NEXT EMPLEADO
          end;
```

11.6.3 *Las órdenes GET PATH y GET NEXT WITHIN PARENT*

Hasta aquí hemos considerado la obtención de registros individuales mediante la orden GET. Sin embargo, cuando necesitamos obtener un registro ubicado en las profundidades de la jerarquía, podemos basar nuestra obtención en una serie de condiciones que deben satisfacer los registros a lo largo de todo el camino jerárquico. Para poder manejar esto, introducimos la orden GET PATH (obtener camino):

```
GET (FIRST | NEXT ) PATH <camino jerárquico> [ WHERE <condición> ]
```

Aquí, <camino jerárquico> es una lista de tipos de registros que comienza con la raíz y sigue un camino por el esquema jerárquico, y <condición> es una expresión booleana que especifica condiciones sobre los tipos de registros individuales que están en dicho camino. Dado que es posible especificar varios tipos de registros, los nombres de los campos llevan como prefijo el nombre de su tipo de registros en <condición>. Por ejemplo, consideremos la siguiente consulta: "Imprimir una lista de los apellidos y fechas de nacimiento de todos los pares empleado-dependiente, donde ambos tengan José como nombre de pila." Esto se muestra en EJ4:

```
EJ4: $GET FIRST PATH EMPLEADO, DEPENDIENTE
     WHERE EMPLEADO.NOMBREP='José'AND DEPENDIENTE.NOMBREDEP='José';
     while DB_STATUS = 0 do
          begin
          writeln (P_EMPLEADO.APELLIDO, P_EMPLEADO.FECHAN,
                    P_DEPENDIENTE.FECHANAC);
     $GET NEXT PATH EMPLEADO, DEPENDIENTE
                    WHERE EMPLEADO.NOMBREP='José' AND
                    DEPENDIENTE.NOMBREDEP='José'
     end;
```

Suponemos que una orden GET PATH obtiene *todos los registros a lo largo del camino especificado* y los coloca en las variables UWA†, y que el último registro del camino se convierte en

†IMS permite especificar que sólo han de obtenerse *algunos* de los registros en el camino.

el registro actual de la base de datos. Además, todos los registros en ese camino se convierten en los *registros actuales de sus respectivos tipos de registros*.

Otro tipo de consulta muy común consiste en buscar todos los registros de un tipo dado que tengan *el mismo registro padre*. En este caso necesitamos la orden GET NEXT WITHIN PARENT (obtener el siguiente dentro del padre), que puede servir para recorrer todos los registros hijo de un registro padre y tiene el siguiente formato:

```
GET NEXT <nombre de tipo de registros hijo>
     WITHIN [ VIRTUAL ] PARENT [ <nombre de tipo de registros padre> ]†
     [ WHERE <condición> ]
```

Esta orden obtiene el siguiente registro del tipo de registros hijo buscando hacia adelante el siguiente registro hijo *propiedad del registro padre actual* a partir del *actual del tipo de registros hijo*. Si no se encuentran más registros hijo, DB_STATUS adquiere un valor distinto de cero para indicar que "no hay más registros del tipo de registros hijo especificado que tengan el mismo padre que el registro padre actual". El <nombre de tipo de registros padre> es *opcional*, y su especificación por omisión es el tipo de registros padre (real) inmediato del <nombre de tipo de registros hijo>. Por ejemplo, si queremos obtener los nombres de todos los proyectos controlados por el departamento de investigación, podemos escribir el segmento de programa que se muestra en EJ5:

```
EJ5:  $GET FIRST PATH DEPARTAMENTO, PROYECTO
           WHERE NOMBRED='Investigación';
           (* esto establece el registro DEPARTAMENTO de
             'Investigación' como padre actual del tipo DEPARTAMENTO
             y obtiene el primer registro PROYECTO hijo bajo ese
             registro DEPARTAMENTO *)
      while DB_STATUS = 0 do
           begin
           writeln (P_PROYECTO.NOMBREPR);
           $GET NEXT PROYECTO WITHIN PARENT
           end;
```

En EJ5 podríamos escribir "WITHIN PARENT DEPARTAMENTO", en vez de sólo "WITHIN PARENT" en la orden GET NEXT, con el mismo resultado, porque DEPARTAMENTO es el tipo de registros padre inmediato de PROYECTO. Sin embargo, si queremos obtener todos los registros propiedad de un padre que *no sea el padre inmediato* —por ejemplo, todos los registros TRABAJADORP propiedad del mismo registro DEPARTAMENTO— deberemos especificar DEPARTAMENTO como tipo de registros padre en la cláusula "WITHIN PARENT".

Adviértase que hay dos métodos principales para establecer explícitamente un registro padre como registro actual:

- Si usamos GET FIRST o GET NEXT, el registro obtenido se convierte en el registro padre actual.
- Si usamos la orden GET PATH, se establece un camino jerárquico de registros padre actuales de sus respectivos tipos de registros. Esto también puede obtener el *primer registro hijo*, como se ilustra en EJ5, para poder emitir órdenes GET NEXT WITHIN PARENT subsecuentes.

†En IMS no hay forma de obtener todos los hijos de un padre virtual como aquí sin definir una vista de la base de datos.

Podemos reescribir EJ4 sin la orden GET PATH si usamos un ciclo para buscar los empleados cuyo NOMBREP sea 'José' y un ciclo anidado con GET NEXT WITHIN PARENT para buscar cualesquier DEPENDIENTEs de cada uno de esos EMPLEADOs con NOMBREDEP = 'José'. Sin embargo, la orden GET PATH nos permite hacer esto de manera más directa y con *un menor número de llamadas* al SGBD.

Con otra variación más de la orden GET podemos localizar el registro padre real o virtual del *registro actual de un tipo de registros hijo especificado*:†

```
GET [ VIRTUAL ] PARENT <nombre de tipo de registros padre>
OF <nombre de tipo de registros hijo>
```

Por ejemplo, si queremos obtener los nombres y horas por semana de todos los empleados que trabajan en 'ProyectoX', podemos usar la orden GET PARENT, como en EJ6:

```
EJ6:  $GET FIRST PATH DEPARTAMENTO, PROYECTO, TRABAJADORP
        WHERE NOMBREPR='ProyectoX'; (* establecer el registro padre y
                                       obtener el primer hijo *)
      while DB_STATUS = 0 do
          begin
          $GET VIRTUAL PARENT EMPLEADO OF TRABAJADORP;
          if DB_STATUS=0 then
              writeln (P_EMPLEADO.APELLIDO, P_EMPLEADO.NOMBREP,
              P_TRABAJADORP.HORAS)
              else writeln ('error--no tiene padre virtual EMPLEADO');
          $GET NEXT TRABAJADORP WITHIN PARENT PROYECTO
          end;
```

Observe que podemos usar una condición WHERE con la orden GET NEXT WITHIN PARENT. Por ejemplo, para obtener los nombres de los empleados que trabajan más de cinco horas a la semana en el 'ProyectoX', podemos cambiar la orden GET NEXT WITHIN PARENT de EJ6 a:

```
$GET NEXT TRABAJADORP WITHIN PARENT PROYECTO WHERE HORAS > '5.0';
```

También deberemos hacer la modificación apropiada a la orden GET FIRST PATH. Así como permitimos el recorrido de un VPHV del hijo al padre, como en EJ6, también podemos recorrerlo del padre al hijo con la siguiente modificación de la orden:

```
GET NEXT <nombre de tipo de registros hijo virtual>
    WITHIN PARENT <nombre de tipo de registros padre virtual>
```

11.6.4 *Cálculo de funciones agregadas*

Las funciones agregadas como CUENTA y PROMEDIO las debe implementar explícitamente el programador, mediante los recursos del lenguaje de programación anfitrión. Por ejemplo,

†En IMS no se cuenta con una contraparte de esta orden que no use lo que en IMS se llama una **base de datos lógica**, que "oculta" esta operación considerando a un padre virtual y a un hijo virtual como un solo registro.

para calcular el número de empleados que trabajan en cada departamento y su salario medio, podemos escribir EJ7:

```
EJ7:  $GET FIRST PATH DEPARTAMENTO, EMPLEADOSD;
      while DB_STATUS = 0 do
          begin
          sal_total:=0;núm_de_emps:=0;writeln(P_DEPARTAMENTO.NOMBRED);
          (* nombre del departamento *)
          while DB_STATUS = 0 do
              begin
              $GET VIRTUAL PARENT EMPLEADO;
              sal_total:=sal_total + conv_sal(P_EMPLEADO.SALARIO);
              núm_de_emps:=núm_de_emps + 1;
              $GET NEXT EMPLEADOSD WITHIN PARENT DEPARTAMENTO
              end;
                  writeln('núm. de empleados =', núm_de_emps, 'salario
                  medio de emps =', sal_total/núm_de_emps);
                  $GET NEXT PATH DEPARTAMENTO, EMPLEADOSD
                  end;
```

Para calcular el salario total y el número de empleados, es preciso declarar variables de programa, así que suponemos que se declararon las variables sal_total:real y núm_de_emps:integer en la cabecera del programa, así como una función conv_sal:real de PASCAL que convierta un valor de salario de cadena a número real.

11.6.5 *Órdenes de HDML para actualización*

Las órdenes de HDML para actualizar una base de datos jerárquica se muestran en la tabla 11.1. La orden INSERT sirve para insertar un registro nuevo. Antes de insertar un registro de un tipo de registros dado deberemos colocar los valores de campos del nuevo registro en la variable de programa apropiada del área de trabajo del usuario. Por ejemplo, suponga que deseamos insertar un nuevo registro EMPLEADO para José F. Silva; podemos usar el segmento de programa EJ8:

```
EJ8:  P_EMPLEADO.NOMBREP := 'José';
      P_EMPLEADO.APELLIDO := 'Silva';
      P_EMPLEADO.INIC := 'F';
      P_EMPLEADO.NSS := '567342739';
      P_EMPLEADO.DIRECCIÓN := 'Ave. Nogal 40, Yautepec, Morelos 55433';
      P_EMPLEADO.FECHAN := '10-ENE-55';
      P_EMPLEADO.SEXO := 'M';
      P_EMPLEADO.SALARIO := '30000.00';
      $INSERT EMPLEADO FROM P_EMPLEADO;
```

La orden INSERT inserta un registro en la base de datos. El registro recién insertado se convierte además en el registro actual de la base de datos, de su esquema jerárquico y de su tipo de registros. Si es un registro raíz, como en EJ8, crea un nuevo árbol de ocurrencia jerárquico con el nuevo registro como raíz. El registro se inserta en la secuencia jerárquica en

el orden especificado por cualesquier campos de ordenamiento que se hayan incluido en la definición del esquema. Por ejemplo, el nuevo registro EMPLEADO de EJ8 se inserta en orden alfabético según su valor combinado de APELLIDO, NOMBREP, de acuerdo con la definición de esquema de la figura 11.11. Si no se especifican campos de ordenamiento en la definición del registro raíz de un esquema jerárquico, el nuevo registro raíz se inserta después del árbol de ocurrencia que contenía el registro de base de datos actual antes de la inserción.

Si se desea insertar un registro hijo, debe hacerse que su padre o uno de sus registros hermanos sea el *registro actual* del esquema jerárquico antes de emitir la orden INSERT. También habría que establecer cualesquier apuntadores a padre virtual antes de insertar el registro. Para hacerlo, necesitamos una orden SET VIRTUAL PARENT (establecer padre virtual), que asigna al campo apuntador de la variable de programa la dirección del registro actual del tipo de registros padre virtual.† El registro se inserta después de encontrar un lugar apropiado para él en la secuencia jerárquica después del registro actual. Por ejemplo, suponga que deseamos relacionar el registro EMPLEADO que insertamos en EJ8 como trabajador de 40 horas por semana con el proyecto cuyo número es 55; podemos usar EJ9:

```
EJ9:  $GET FIRST EMPLEADO WHERE NSS='567342739'; (* buscar padre virtual *)
      if DB_STATUS=0 then
            begin
            P_TRABAJADORP.APUNTT := SET VIRTUAL PARENT; (* apuntador de
            padre virtual al registro actual *)
            P_TRABAJADORP.HORAS := '40.0';
            $GET FIRST PROYECTO WHERE NÚMEROP=55; (* hacer al
            padre (real) el registro actual *)
            if DB_STATUS=0 then $INSERT TRABAJADORP FROM P_TRABAJADORP;
            end;
```

Para eliminar un registro de la base de datos, primero hacemos que sea el registro actual y luego emitimos la orden DELETE. Con la orden GET HOLD se convierte el registro en el registro actual, donde la palabra reservada HOLD (mantener) le indica al SGBD que el programa eliminará o actualizará el registro recién leído. Por ejemplo, si deseamos eliminar todos los empleados de sexo masculino, podemos usar EJ10, que también lista los nombres de los empleados eliminados *antes* de eliminar sus registros:

```
EJ10: $GET HOLD FIRST EMPLEADO WHERE SEXO='M';
      while DB_STATUS=0 do
            begin
            writeln (P_EMPLEADO.APELLIDO, P_EMPLEADO.NOMBREP);
            $DELETE EMPLEADO;
            $GET HOLD NEXT EMPLEADO WHERE SEXO='M';
            end;
```

Observe que la eliminación de un registro implica eliminar automáticamente todos sus registros descendientes, esto es, todos los registros de su subárbol. Sin embargo, los registros hijo virtuales que están en otras jerarquías no se eliminan. De hecho, antes de eliminar un registro, el SGBD debe asegurarse de que ningún registro hijo virtual apunte a él.

†La acción de SET VIRTUAL PARENT se efectúa implícitamente en IMS cuando se inserta un registro lógico que contiene al padre virtual en su definición.

Después de una orden DELETE ejecutada con éxito, el registro actual se convierte en una "posición vacía" en la secuencia jerárquica correspondiente al registro recién eliminado. Las operaciones subsecuentes partirán de esa posición.

A fin de modificar valores de los campos de un registro, seguiremos estos pasos:

1. Hacer que el registro por modificar sea el registro actual, obtenerlo y colocarlo en la variable de programa UWA correspondiente mediante la orden GET HOLD.
2. Modificar los campos deseados en la variable de programa UWA.
3. Emitir la orden **REPLACE** (reemplazar).

Por ejemplo, si queremos conceder a todos los empleados del departamento de investigación un aumento salarial del 10%, podremos usar el programa que se muestra en EJ11:

```
EJ11: $GET FIRST PATH DEPARTAMENTO, EMPLEADOSD
            WHERE NOMBRED='Investigación';
      while DB_STATUS=0 do
            begin
            $GET HOLD VIRTUAL PARENT EMPLEADO OF EMPLEADOSD;
            P_EMPLEADO.SALARIO := P_EMPLEADO.SALARIO * 1.1;
            $REPLACE EMPLEADO FROM P_EMPLEADO;
            $GET NEXT EMPLEADOSD WITHIN PARENT DEPARTAMENTO
            end;
```

11.7 Panorama del sistema de bases de datos jerárquicas IMS★

11.7.1 *Introducción*

En esta sección examinaremos un importante sistema jerárquico: el Information Management System (sistema de gestión de información) o IMS. Aunque en lo esencial IMS implementa el modelo de datos jerárquico hipotético que describimos en secciones anteriores de este capítulo, muchas características son exclusivas de este complejo sistema. Destacaremos la arquitectura y los tipos especiales de procesamiento de vistas y estructuras de almacenamiento de IMS, y compararemos DL/1, el lenguaje de datos de IMS, con los HDDL y HDML que analizamos antes.

IMS fue uno de los primeros SGBD, y se ha colocado en el mercado como el sistema dominante para el manejo de sistemas de contabilidad y de inventarios a gran escala. Los manuales de IBM se refieren al producto completo como IMS/VS (*Virtual Storage*: almacenamiento virtual) y, por lo regular, este producto se instala bajo el sistema operativo MVS. IMS DB/DC es el término aplicado a las instalaciones que utilizan los subsistemas propios del producto para manejar la base de datos física (DB) y para proporcionar comunicaciones de datos (DC).

No obstantte, existen otras versiones importantes que sólo manejan el lenguaje de datos de IMS (DL/1). Estas configuraciones se pueden implementar bajo MVS, pero también pueden usar el sistema operativo DOS/VSE. Estos sistemas emiten sus llamadas a archivos VSAM y emplean el Customer Information Control System (CICS: sistema de control de

información de clientes) de IBM para la comunicación de datos. Aquí se sacrifica el manejo de más características en aras de la sencillez y de una mayor productividad.

IBM introdujo el producto original IMS/360 Version 1 en 1968, después de un proyecto de desarrollo conjunto con North American Rockwell. Habrían de seguir varias revisiones importantes de IMS, que han incorporado o contemplado señalados avances tecnológicos: modernas redes de comunicaciones, acceso directo a registros ("camino rápido" aumentado) e índices secundarios, entre otros. La creación de IMS representa ahora varios miles de años-hombre de esfuerzo, que en gran medida ha sido impulsado por las necesidades de una comunidad de usuarios que no duda en expresar sus deseos.

IMS no cuenta con un lenguaje de consulta integrado, lo cual puede considerarse una seria deficiencia. Casi desde el principio aparecieron respuestas parciales a esta situación, con el IQF (*Interactive Query Facility*: recurso de consulta interactiva) de IBM y otros productos añadidos que venden los proveedores o que crean los usuarios. Hoy día, una solución común y de alta flexibilidad consiste en transferir información de la base de datos IMS, que suele ser enorme, a un sistema relacional aparte. Después, habiendo pasado los datos resumidos pertinentes a un microcomputador o al sistema SQL/DS o DB2 del computador central, las entidades corporativas individuales pueden ejecutar sus propias funciones de sistema de información.

Se han comercializado varias versiones de IMS que trabajan con diversos sistemas operativos de IBM, de los que podríamos mencionar (entre los sistemas recientes) OS/VS1, OS/VS2, MVS, MVS/XA y ESA. El sistema viene con varias opciones. IMS se ejecuta bajo diferentes versiones en la familia de computadores IBM 370 y 30XX. El lenguaje de definición y manipulación de datos de IMS es Data Language One (DL/1). Los programas de aplicación escritos en COBOL, PL/1, FORTRAN y BAL (Basic Assembly Language: lenguaje ensamblador básico) se comunican con DL/1.

System 2000 (S2K) es otro sistema jerárquico muy utilizado que sigue una versión diferente del modelo de datos jerárquico. Opera en una amplia gama de sistemas, incluidos los modelos IBM 360/370, 43XX y 30XX, así como en equipos UNIVAC, CDC y CYBER. System S2K se puede configurar con opciones como un lenguaje de consulta no orientado a procedimientos para no programadores, una interfaz de lenguaje por procedimientos para COBOL, PL/1 y FORTRAN, un recurso de procesamiento de archivos secuenciales y un supervisor de teleproceso. En el resto de esta sección describiremos diversos aspectos de IMS.

11.7.2 *Arquitectura básica de* IMS

La organización interna de IMS puede describirse en términos de diversas capas de definiciones y correspondencias. IMS usa su propia terminología, que en ocasiones produce confusión o tiene conotaciones equivocadas. Una jerarquía almacenada en IMS se llama **base de datos física** (PDB: *physical database*). En una instalación determinada, la información de la base de datos comprende varias bases de datos físicas. Cada una de éstas tiene una definición de datos o esquema escrito en DL/1. IMS llama a esta definición DBD (*database description*: descripción de base de datos). La forma compilada de una DBD se almacena internamente; incluye información sobre la correspondencia entre la definición de la base de datos y el almacenamiento, y sobre los métodos de acceso aplicables.

IMS cuenta con un recurso de vistas que es bastante complejo. Una vista puede definirse escogiendo parte de una base de datos física o partes de varias bases de datos físicas y entrelazándolas para formar una nueva jerarquía. Llamaremos a éstas vistas tipo 1 y tipo 2, respectivamente. (La nomenclatura de tipos es nuestra.)

Una vista de tipo 1 es una subjerarquía y se define mediante un **bloque de comunicación de programa** o PCB (*program communication block*). Una vista de tipo 2 debe definirse en DL/1 en términos de una DBD *lógica*. La estructura resultante se denomina **base de datos lógica** (LDB: *logical database*). Las bases de datos físicas y lógicas se estudiarán en la sección 11.7.3.

Los programas de aplicación de los usuarios necesitan tener acceso a los datos de varias bases de datos físicas aisladas o de vistas tipo 1 o tipo 2. En los sistemas de transacciones en línea de alto volumen se usan módulos reentrantes de acceso a datos. Todas las descripciones de datos que necesita una aplicación se empacan en un **bloque de especificación de programa** o PSB (*program specification block*). Un PSB contiene diferentes porciones descriptivas, que corresponden a definiciones de vistas tipo 1 o tipo 2. Estas porciones se almacenan como bloques de comunicación de programa. *Cada aplicación debe tener un PSB distinto, aunque puede ser idéntico a otro PSB.* El programa de aplicación en COBOL, PL/1, FORTRAN o BAL invoca a DL/1 mediante una llamada para que IMS atienda una operación de obtención o actualización. El sistema IMS, a su vez, se comunica con el usuario a través del PCB, el cual se define en el programa de aplicación como un área direccionable a través de un apuntador que se pasa al programa. La información de estado actual se envía al PCB. IMS maneja principalmente cinco métodos de acceso, HSAM, HISAM, HDAM, HIDAM y MSDB, que a su vez usan los métodos de acceso integrados del sistema operativo para gestionar diversos archivos.

La figura 11.15 muestra cómo dos aplicaciones, VENTAS y CONTABILIDAD, podrían compartir bases de datos físicas en IMS a través de los DBD, PCB y PSB.

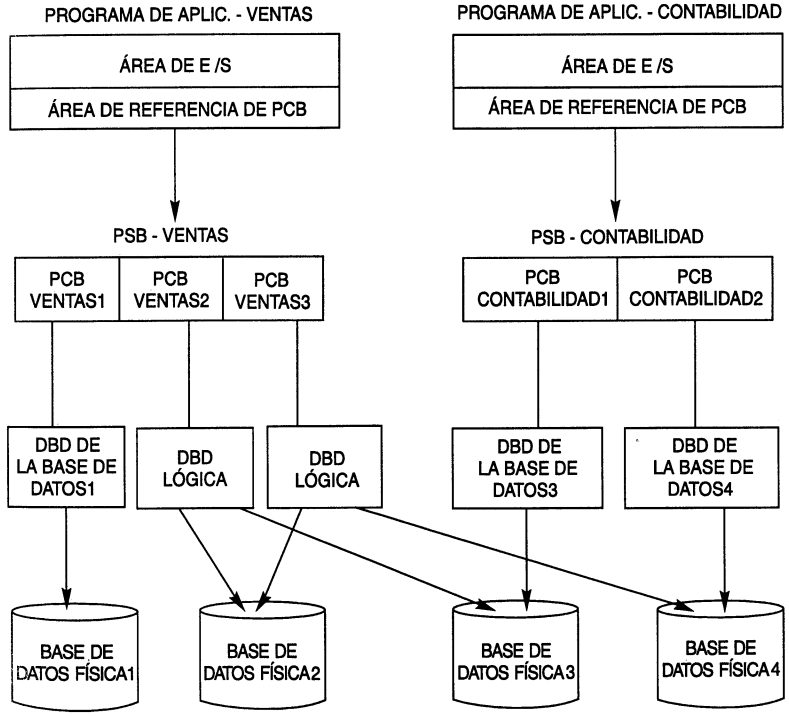


Figura 11.15 Compartimiento de datos entre dos aplicaciones IMS.

11.7.3 *Organización lógica de los datos en* IMS

En IMS, los registros se llaman segmentos, y los vínculos se caracterizan como físicos y lógicos (en vez de reales y virtuales). La tabla 11.2 muestra referencias cruzadas entre nuestra terminología, la de IMS y la de System 2000.

Bases de datos físicas. El término base de datos física (PDB) de IMS se refiere a la jerarquía que se almacena realmente. Se define con una DBD física en el lenguaje DL/1. La figura 11.16 muestra la definición de una base de datos física que corresponde a la jerarquía de la figura 11.5. Contiene seis tipos de segmentos, cada uno de los cuales puede tener un número arbitrario de ocurrencias en la base de datos. Para el esquema de la figura 11.10 tendríamos que usar dos definiciones de bases de datos físicas. Más adelante se definirían bases de datos lógicas apropiadas a partir de estas bases de datos físicas. La definición de los vínculos padre hijo virtuales que aparecen en la figura 11.10 está incluida en estas dos DBD y es bastante complicada.

Destacamos varios aspectos importantes sobre la definición de base de datos:

- La descripción de la base de datos está escrita en términos de las macros DBD, SEGM, FIELD, DBDGEN, FINISH y END. La macro SEGM define un segmento, la macro FIELD define un campo, DBDGEN genera la DBD y las macros FINISH y END terminan la descripción.

Tabla 11.2 Terminología para el modelo de datos jerárquico

Modelo jerárquico	Término de IMS	Término de System 2000
1. Tipo de registros	Tipo de segmentos	Registro de grupo repetitivo
2. Ocurrencia de registro	Ocurrencia de segmento	Ocurrencia de registro
3. Campo o elemento de información	Campo	Elemento de información
4. Campo de secuencia como clave	Campo de secuencia	Clave
5. Tipo de vínculos padre-hijo	Tipo de vínculos padre-hijo físicos	Vínculo de esquema jerárquico
6. Tipo de vínculos padre-hijo virtuales	Tipo de vínculos padre-hijo lógicos	No hay, excepto en el momento de la ejecución
7. Esquema de base de datos jerárquica	Definición de base de datos física o lógica (hecha en la DBD)	Árbol de esquema
8. Raíz de jerarquía	Segmento raíz	Registro raíz
9. Árbol de ocurrencia de una jerarquía	Registro de base de datos física	Árbol de datos
10. Secuencia jerárquica de registros	Secuencia jerárquica	No hay un término especial
11. Tipo de registros apuntador	Tipo de segmentos apuntador	No hay un concepto similar

- Cada macro usa ciertas palabras reservadas. La estructura jerárquica lógica de la base de datos se define en virtud de las especificaciones "PARENT =" (padre =) de los segmentos.
- El orden de ocurrencia de los enunciados SEGM es la forma de ordenar los segmentos dentro del esquema lógico. Este ordenamiento de arriba a abajo y de izquierda a derecha es significativo; si alteramos este orden tendremos una base de datos física *diferente*.
- Un *campo de secuencia* (opcional) designa un campo dentro de un tipo de segmentos según el cual pueden ordenarse sus ocurrencias. El valor específico del campo de secuencia es la clave de esa ocurrencia de segmento.
- Los campos de secuencia pueden ser únicos (por omisión) o no únicos. Para designar un campo de secuencia no único se utiliza una M (de múltiple) en la definición FIELD, como aquí:

```
FIELD NAME = (NOMBRECOMP, SEQ.M), ...
```

```
1  DBD  NAME   = COMPAÑÍA
2  SEGM NAME   = DEPARTAMENTO, BYTES = 28
3  FIELD NAME  = NOMBRED, BYTES = 10, START = 1
4  FIELD NAME  = (NÚMEROD, SEQ), BYTES = 6, START = 11
5  FIELD NAME  = NOMBREGTE, BYTES = 3, START = 17
6  FIELD NAME  = FECHAINICGTE, BYTES = 9, START = 20

7  SEGM NAME   = EMPLEADO, PARENT = DEPARTAMENTO, BYTES = 79
8  FIELD NAME  = NOMBRE, BYTES = 31, START = 1
9  FIELD NAME  = (NSS, SEQ), BYTES = 9, START = 32
10 FIELD NAME  = FECHAN, BYTES = 9, START = 41
11 FIELD NAME  = DIRECCIÓN, BYTES = 30, START = 50

12 SEGM NAME   = DEPENDIENTE, PARENT = EMPLEADO, BYTES = 25
13 FIELD NAME  = (NOM_DEP, SEQ), BYTES = 15, START = 1
14 FIELD NAME  = SEXO, BYTES = 1, START = 16
15 FIELD NAME  = FECHANAC, BYTES = 9, START = 17

16 SEGM NAME   = SUPERVISADO, PARENT = EMPLEADO, BYTES = 24
17 FIELD NAME  = NOMBRE, BYTES = 15, START = 1
18 FIELD NAME  = NSS, BYTES = 9, START = 16

19 SEGM NAME   = PROYECTO, PARENT = DEPARTAMENTO, BYTES = 16
20 FIELD NAME  = NOMBREPR, BYTES = 10, START = 1
21 FIELD NAME  = (NÚMEROP, SEQ), BYTES = 6, START = 11

22 SEGM NAME   = TRABAJADOR, PARENT = PROYECTO, BYTES = 26
23 FIELD NAME  = NOMBRE, BYTES = 15, START = 1
24 FIELD NAME  = (NSS, SEQ), BYTES = 9, START = 16
25 FIELD NAME  = HORAS, BYTES = 2, START = 25
26 DBDGEN
27 FINISH
28 END
```

Figura 11.16 Definición de base de datos física correspondiente a la jerarquía de la figura 11.5.

- Se requiere un campo de secuencia único para el segmento raíz si la base de datos se almacena empleando HISAM o HIDAM (véase la Sec. 11.7.5), ya que constituye una clave de índice para el índice primario.
- Las combinaciones de dos o más campos se reconocen como campos nuevos. Esto permite tratar una combinación de campos como clave compuesta. Por ejemplo, podríamos dar un nuevo nombre, digamos NOMBREEC, a NOMBREESTADO y NOMBRECIUDAD juntos, y definirlo como campo de secuencia.

Un árbol de ocurrencia en nuestra terminología se llama **registro de base de datos física** en IMS. La forma linealizada de un árbol de ocurrencia dentro de un registro de base de datos física se produce mediante un *recorrido en preorden* de las ocurrencias de segmentos (véase la Sec. 11.1.4). La secuencia de segmentos desde cualquier segmento hasta la raíz (que se obtiene pasando por una serie de segmentos padre sucesivos) se denomina **camino jerárquico** del segmento. Una concatenación de claves (incluidos los códigos de tipo de segmentos) a lo largo de este camino se llama **clave de secuencia jerárquica** de ese segmento. La clave de secuencia jerárquica de una ocurrencia de DEPENDIENTE en un registro de base de datos física para la base de datos de la figura 11.5 podría ser ésta:

1 | '000005' | 2 | '369278157' | 4 | 'JOSÉ... '.

Aquí, 1, 2 y 4 son, respectivamente, los códigos de tipo de segmentos de DEPARTAMENTO, EMPLEADO y DEPENDIENTE, mismos que IMS asigna automáticamente, y '000005', '369278157' y 'JOSÉ...' son claves de secuencia que ascienden por el camino jerárquico hasta esa ocurrencia de segmento de JOSÉ.

Los registros de base de datos física de una base de datos IMS ocurren en orden según la clave de su segmento raíz. Dentro de un registro de base de datos física, los segmentos ocurren en orden ascendente según su clave de secuencia jerárquica.

Vistas tipo 1 en IMS — Subconjuntos de bases de datos físicas. IMS permite construir dos clases de vistas, o "bases de datos lógicas"† (término de IMS), a partir de las bases de datos físicas. Para facilitar la referencia les llamaremos bases de datos lógicas tipo 1 y tipo 2, o vistas tipo 1 y tipo 2, que es nuestra propia nomenclatura. Por cierto, IMS da lugar a una confusión adicional porque sólo las vistas tipo 2 se definen realmente con una definición de base de datos lógica; las vistas tipo 1 se definen con un PCB (bloque de comunicación del programa).

Un esquema de base de datos lógica tipo 1 define una *subjerarquía* de un esquema de base de datos física obedeciendo estas reglas:

1. El tipo de segmentos raíz debe ser parte de la vista.
2. Los tipos de segmentos no raíz pueden omitirse.
3. Si se omite un tipo de segmentos, deberán omitirse todos sus tipos de segmentos hijo.
4. De los segmentos incluidos, puede omitirse cualquier tipo de campos.

Por ejemplo, podemos definir un gran número de vistas tipo 1 para la base de datos de la figura 11.5. Dos vistas tipo 1 válidas se muestran en la figura 11.17(a). Están orientadas a dos aplicaciones distintas: una se ocupa de los dependientes y la otra de los empleados que trabajan en proyectos.

†IMS usa el término *base de datos lógica* sin mucho rigor, con dos significados que corresponden a los dos tipos de vistas. Sin embargo, una base de datos lógica (LDB) sólo está definida para la jerarquía virtual o vista tipo 2.

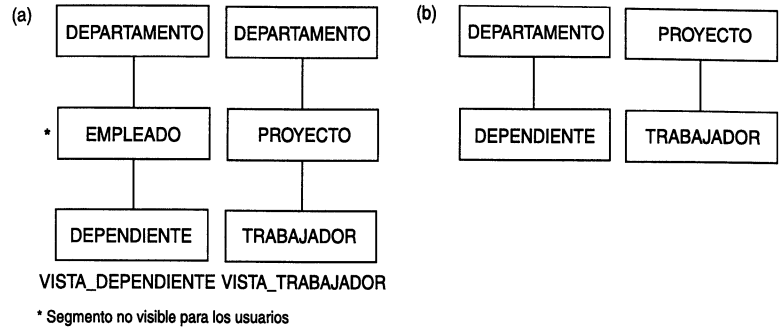


Figura 11.17 Vistas IMS tipo 1 sobre la base de datos de la figura 11.5. (a) Dos vistas válidas. (b) Dos vistas no válidas.

En la figura 11.17(b) se muestran otras dos subjerarquías. En la primera, el segmento EMPLEADO se omite pero se incluye su segmento hijo. Esto viola la regla 3 de la lista anterior. En la segunda subjerarquía, se omite el segmento raíz DEPARTAMENTO, lo que viola la regla 1. Por tanto, ninguna de estas subjerarquías es una vista válida en IMS. Todas las jerarquías tipo 1 se definen mediante un PCB. No describiremos aquí la sintaxis de los PCB.

Este recurso de vistas logra el objetivo usual de permitir acceso selectivo sólo a la parte pertinente de una base de datos, y ofrece un cierto grado de seguridad. Cuando la especificación PROCOPT permite actualizaciones, con el cambio correspondiente es posible actualizar el registro físico "base". Esto está gobernado por un conjunto complejo de reglas en IMS y puede dar lugar a inconsistencias si no se efectúa correctamente. Un PSB (bloque de especificación de programa) para una aplicación dada puede incluir varios PCB que correspondan a varias vistas tipo 1.

Vistas IMS tipo 2 sobre múltiples bases de datos físicas. Este recurso de IMS es un verdadero recurso de vistas en cuanto a que permite la creación de vistas que son jerarquías virtuales. Una **jerarquía virtual** consta de segmentos, algunos de los cuales se conectan mediante *vínculos padre-hijo lógicos* (término de IMS; nosotros los llamamos vínculos padre-hijo virtuales —VPHV— en la sección 11.2). Mediante el establecimiento de vínculos lógicos entre segmentos de diferentes bases de datos físicas, podemos crear una red compleja. El recurso de vistas tipo 2 nos permite extraer cualquier jerarquía de una red así. En la figura 11.18 mostramos jerarquías virtuales basadas en las jerarquías de la figura 11.10. Las vistas tipo 2 deben definirse explícitamente en IMS como bases de datos lógicas (LDB) usando la macro DBD y con ACCESS = LOGICAL.

Son varias las reglas que gobiernan los vínculos padre-hijo lógicos y la construcción de bases de datos lógicas a partir de bases de datos físicas. Entre las más importantes están:

1. La raíz de una LDB debe ser la raíz de alguna PDB.
2. Un segmento hijo lógico debe tener un padre físico y sólo un padre lógico. En consecuencia, un segmento raíz *no puede* ser un segmento hijo lógico.
3. Un hijo físico de un padre lógico puede aparecer como dependiente de un segmento concatenado (hijo lógico/padre lógico) en la LDB. Gracias a este recurso, podemos

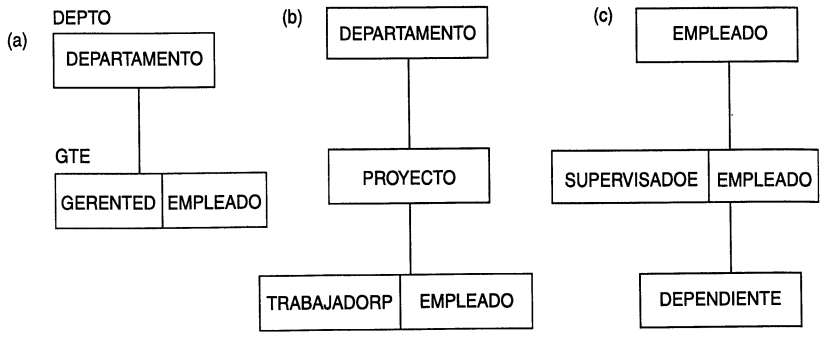


Figura 11.18 Vistas IMS tipo 2 sobre la base de datos de la figura 11.10.
(a) Vista_Gerente. (b) Vista_Proyecto. (c) Vista_Dependiente.

crear una base de datos lógica como la de la figura 11.18 a partir de la figura 11.10; esto demuestra el uso de un tipo de segmentos EMPLEADO dos veces (una como supervisor y otra como supervisado) en la misma base de datos lógica. Un resultado es que seguir la pista a los indicadores de actualidad de los segmentos se hace aún más difícil de lo indicado en la sección 11.6.1. Ésta es una característica muy potente de IMS y expande considerablemente el alcance de la generación de nuevas jerarquías.

4. Un tipo de segmentos padre lógico puede tener múltiples tipos de segmentos hijo lógicos. Esto ya se vio con el segmento EMPLEADO en la figura 11.10.

Vistas en IMS vs. vistas en sistemas relacionales. Examinamos dos tipos de definiciones de vistas o recursos de esquemas externos en IMS. El tipo 1 abarca una sola jerarquía, y el tipo 2, varias. Comparemos este recurso de vistas con las vistas en los sistemas relacionales:

1. Las vistas relacionales no deben contemplarse en el momento de definir un esquema conceptual o un conjunto de relaciones base. En contraste, la definición de las PDB está determinada por las LDB que necesitan usarlas. Por tanto, las vistas IMS tipo 2 no son esquemas puramente externos; influyen en la definición del esquema conceptual y lo determinan. Así pues, el espíritu de la arquitectura de tres esquemas (véase la Sec. 2.2.1) *no* se mantiene por completo.
2. Las vistas relacionales no suponen ninguna estructura de acceso físico para manejar las vistas. En cambio, las vistas IMS tipo 2 requieren la definición explícita de apuntadores para vincular segmentos de múltiples PDB. Las LDB factibles están limitadas por los tipos de apuntadores declarados en la o las bases de datos físicas.
3. La definición de una vista tipo 1 es obligatoria para que una aplicación pueda tener acceso a una base de datos física (o lógica). Incluso si una aplicación tiene acceso a toda la base de datos física, habrá que definir un PCB para ella (vista tipo 1). De hecho, se requiere un PCB sobre una LDB para tener acceso a ella. En los sistemas relacionales no existe esta clase de requisitos.

El recurso de vistas tipo 2 es una característica útil que extiende las capacidades de IMS en estos sentidos:

- Hace posible un **recurso de red limitado** al permitir que dos segmentos tengan un vínculo M:*N* a través de un segmento apuntador hijo común. Los vínculos *N*-arios con $N > 2$ *no* son posibles, a diferencia de lo que sucede en el modelo de red.
- Reduce el almacenamiento redundante de datos. Por ello, las actualizaciones pueden guardarse.
- Lo más importante es que permite a los usuarios ver los datos desde diferentes perspectivas jerárquicas, además de las jerarquías de base de datos física rígidamente definidas. Esto se hace combinando segmentos de múltiples jerarquías existentes.

Desafortunadamente, las definiciones de bases de datos físicas y lógicas, los diferentes tipos de apareamiento de segmentos —físicos y virtuales— en "vínculos bidireccionales" y los complicados procedimientos de carga para las bases de datos lógicas hacen que las vistas tipo 2 sean una característica muy compleja de IMS. Hemos omitido gran cantidad de detalles en nuestro análisis. Al parecer, algunas instalaciones IMS se las arreglan sin bases de datos lógicas y se conforman con depender por completo de las bases de datos físicas.

11.7.4 *Manipulación de datos en* IMS

Las operaciones de manipulación de datos en IMS son muy parecidas a las operaciones de HDML de la sección 11.6. DL/1 contiene el lenguaje de manipulación de datos (DML) de IMS además del lenguaje de definición de datos (DDL). Aquí no describiremos con detalle la sintaxis del lenguaje; sólo mostraremos cómo las aplicaciones DL/1 se comunican con IMS y señalaremos unas cuantas características especiales de DL/1. Las llamadas a DL/1 se incorporan en un programa de aplicación IMS escrito en COBOL, PL/1, lenguaje ensamblador básico de System 360/370 o FORTRAN. Dichas llamadas tienen la siguiente sintaxis:

```
CALL <nombre de procedimiento> (<lista de parámetros>).
```

El nombre del procedimiento que se invoca varía dependiendo del lenguaje en el que está escrito el programa de aplicación. Un programa en PL/1 debe usar el nombre PLITDLI (la "T" proviene de la palabra *to*, "a"), que es fijo. Consideremos la siguiente consulta: "obtener una lista de los dependientes nacidos después del 1o. de enero de 1980 para los empleados del departamento número 4". Una llamada a DL1 se codificaría así:

```
CALL PLITDLI (SIX, GU, PCB_1, DEPND_IO_AREA, D_SSA, EMPL_SSA, DEPND_SSA)
```

Esta llamada aparece en el programa de aplicación que se supone está en PL1. La lista de parámetros se interpreta como sigue:

- SIX se refiere a una variable que contiene la cadena 'SIX' (seis). Indica el número de parámetros que quedan en la lista. Distintas consultas pueden tener diferentes números de parámetros.
- GU representa una variable que contiene la cadena 'GU' y que indica la operación que se realizará; en este caso, "get unique".
- PCB_1 es el nombre de la estructura, definida en el programa en PL/1, que actúa como máscara para direccionar un área llamada bloque de comunicación del programa

(PCB). Es el área común para la transferencia de información entre IMS y el programa de aplicación. Entre otras cosas, incluye un indicador de nivel de la jerarquía, las opciones de procesamiento vigentes, el nombre del segmento actual, la clave de secuencia jerárquica actual y el número de segmentos confidenciales para la definición del PCB correspondiente.

- DEPND_IO_AREA es un área de entrada/salida de 25 bytes reservada en el programa para recibir todo el segmento DEPENDIENTE.
- D_SSA, EMPL_SSA y DEPND_SSA son argumentos para búsqueda de segmentos (*segment search arguments*), uno por consulta. Representan cadenas que contienen variables para especificar las condiciones de búsqueda. En nuestro ejemplo las tres cadenas contendrían 'DEPARTAMENTO(NÚMEROD = '000004')', 'EMPLEADO' y 'DEPENDIENTE(FECHANAC > 'ENE-01-1980')', respectivamente.

En el lenguaje HDML de la sección 11.6, el efecto de la llamada anterior sería ejecutar la siguiente consulta:

```
GET FIRST PATH DEPARTAMENTO, EMPLEADO, DEPENDIENTE
WHERE NÚMEROD = '000004' AND DEPENDIENTE.FECHANAC > 'ENE-01-1980'
```

La única diferencia es que en HDML supusimos que esto obtendría datos para cada uno de los segmentos; en IMS, con la orden Get Unique, *sólo* se carga en la memoria el segmento DEPENDIENTE (el nodo terminal del camino).

Interfaz CALL vs. interfaz de lenguaje de consulta incorporado. El ejemplo anterior ilustra el empleo de una 'interfaz CALL' con parámetros entre un lenguaje de alto nivel como PL/1 y el SGBD. Vale la pena contrastar esto con la interfaz de lenguaje de consulta incorporado que ejemplifica el empleo de SQL en los sistemas relacionales. Las ventajas de una interfaz CALL son:

1. El compilador del lenguaje anfitrión no sufre modificación alguna, por lo que no es necesaria una precompilación.
2. El programa de aplicación tiene un aspecto homogéneo; no interviene ninguna sintaxis ajena.

Las principales desventajas son:

1. Es fácil intercambiar u omitir los parámetros posicionales de una llamada.
2. Si examinamos una instrucción CALL, es imposible juzgar las operaciones de obtención o actualización de datos incorporadas. Esto hace que los programas de aplicación sean de difícil lectura.
3. No hay verificación semántica de los parámetros de una llamada; los errores pueden surgir posteriormente durante la ejecución sin ser detectados durante la compilación.

En nuestra opinión la interfaz CALL puede ser más conveniente para el implementador, pero no es deseable para la creación de aplicaciones.

La tabla 11.3 resume la correspondencia entre las operaciones propuestas en HDML y las que existen en DL/1. Las operaciones de DL/1 se invocan empleando el recurso CALL que

acabamos de describir. En la sección 11.6 adoptamos una notación distinta, en la que las órdenes de HDML estaban incorporadas en programas en PASCAL y las condiciones de búsqueda se escribían en una cláusula WHERE. Esta distinción debe tenerse presente al leer la tabla. A continuación ofrecemos unas cuantas explicaciones adicionales que corresponden a las notas de la tabla 11.3.

- *Nota 1:* Cada vez que se procesa una jerarquía, en general, IMS requiere que la primera orden sea Get Unique (GU), la cual debe direccionar un camino jerárquico en la jerarquía, comenzando por la raíz. No es posible tener acceso directo a segmentos dentro de la jerarquía. (Hay excepciones pero rebasan el alcance de este análisis.)
- *Nota 2:* En DL/1 se usa Get Unique para dar cuenta de GET FIRST y también de GET FIRST PATH de HDML. En el caso de los ejemplos 1, 2 y 3 de la sección 11.6.2, GU de DL/1 funcionaría exactamente igual que GET FIRST de HDML. A continuación mostramos el ejemplo 4 de la sección 11.6.3 en DL/1 más un lenguaje anfitrión (en una pseudosintaxis *sin* codificar como CALL exacta). Esta consulta obtiene los pares empleado-dependiente en los que ambos tienen el nombre de pila José. Los argumentos para búsqueda de segmentos (p. ej., "EMPLEADO*D (NOMBREP = 'José')") se muestran junto a la operación por conveniencia notacional.

```
GU EMPLEADO*D (NOMBRP = 'José')
 DEPENDIENTE (NOMBREDEP = 'José')
 while DB_STATUS = 'se encontró segmento' do
          begin
             WRITE EMPLEADO APELLIDO, EMPLEADO FECHAN,
                   DEPENDIENTE FECHANAC
             GN EMPLEADO*D (NOMBREP = 'José')
                DEPENDIENTE (NOMBREDEP = 'José')
          end;
```

La *D (de "datos") del ejemplo anterior es un **código de orden**. Observe que Get Next con *D produce el mismo efecto que Get Next Path en HDML. En DL/1 sería posible codificar este ejemplo sin usar *D; en tal caso, en vez de todo el camino, sólo se obtendría el segmento terminal (DEPENDIENTE, en el ejemplo anterior).

- *Nota 3:* No hay órdenes especiales en DL/1 para procesar vínculos padre-hijo virtuales ("lógicos" en IMS), porque los vínculos virtuales *no se pueden procesar directamente sin definir una base de datos lógica.*

Cabe hacer unas cuantas observaciones adicionales al comparar DL/1 con HDML. En el ejemplo 10 de la sección 11.6.5 se ilustró la opción de retener (*hold*) para HDML. Se aplica a todas las formas de la operación GET en IMS. Los ciclos while...do...end que se muestran en los ejemplos en PASCAL de la sección 11.6 estaban bajo el control de un código DB_STATUS. Los ciclos se deben codificar explícitamente de la misma manera en el lenguaje anfitrión de IMS. Como parte del área de PCB se dispone de un código de estado.

IMS cuenta con un código de orden *F con el que una aplicación puede realizar una búsqueda en la jerarquía para determinar si se satisface o no una condición y luego regresar a un segmento previamente nombrado dentro del mismo registro de base de datos física y obtener datos. El código de orden *V sirve para ubicar el proceso de obtención

Tabla 11.3 Operaciones propuestas para HDML (Sec. 11.6) y las existentes en DL/1 de IMS

Operación de HDML	**Operación de DL/1**	**Significado**
Get First (GF)	Get Unique (GU)	Obtener la primera ocurrencia de un registro especificado (*véase la Nota 1*)
Get Next (GN)	Get Next (GN)	Obtener la siguiente ocurrencia de un registro especificado
Get First Path	Get Unique (GU) más código de orden *D	Obtener la primera ocurrencia de todos los registros en un camino jerárquico (*véase la nota 2*)
Get Next Path	Get Next within Parent (GNP) o Get Next (GN)	Obtener la siguiente ocurrencia de un camino jerárquico especificado (*véase la nota 3*)
Get Next within Parent	Get Next within Parent (GNP)	Obtener la siguiente ocurrencia de hijo para la ocurrencia de padre actual
Get Next within Virtual Parent	Ninguna operación en especial	Obtener la siguiente ocurrencia de hijo para la ocurrencia de padre virtual actual (*véase la nota 3*)
INSERT	INSERT (ISRT)	Insertar una nueva ocurrencia de registro
DELETE	DELETE (DLET)	Eliminar la vieja ocurrencia de registro
REPLACE	REPLACE (REPL)	Reemplazar la ocurrencia de registro actual por una nueva ocurrencia
Get with Hold	Get Unique with Hold (GHU) Get Next with Hold (GNH) Get Next within Parent with Hold (GHNP)	Realizar una operación GET correspondiente reteniendo el registro para poderlo reemplazar o eliminar posteriormente.

en el actual de un tipo de segmentos específico (véase la Sec. 11.6.2). Con estos códigos de orden, es posible alterar la dirección normal (hacia adelante) de procesamiento dentro de las secuencias jerárquicas linealizadas.

Un programa de aplicación IMS puede abrir varios PCB o vistas tipo 1 y procesarlos simultáneamente. El sistema mantiene los indicadores de actualidad de tipos de segmentos dentro de cada PCB. Cada CALL incluye un parámetro que hace referencia a un PCB específico.

11.7.5 *Almacenamiento de bases de datos en IMS*

Repasaremos los diferentes tipos de organizaciones de archivo disponibles para almacenar bases de datos físicas en IMS, sin entrar en detalles. En IMS a éstas se les llama métodos de acceso. Comparado con la mayoría de los SGBD, IMS provee un surtido mucho más amplio de métodos de acceso. Los mencionaremos a todos aquí:

- Cada base de datos física en IMS es una base de datos almacenada. Las bases de datos lógicas son bases de datos jerárquicas virtuales que se pueden visualizar como tales, pero en almacenamiento no representan datos independientes. Las bases de datos lógicas consisten en las bases de datos físicas más enlaces provistos por estructuras de apuntadores.
- Cada segmento almacenado contiene campos de datos almacenados más un prefijo (invisible para los programas del usuario) que contiene un código de tipo de segmentos, apuntadores, una bandera de eliminación y otra información de control.
- Sea cual sea el método de acceso, una base de datos almacenada siempre se almacena como una secuencia de árboles de ocurrencia, llamados **registros de base de datos física**, donde cada árbol de ocurrencia contiene una secuencia de segmentos en preorden (véanse las Secs.11.1.3 y 11.1.4). Es propiedad de un segmento raíz específico. Por brevedad, en ocasiones llamaremos sólo **árbol** a un árbol de ocurrencia.
- Los diversos métodos de acceso de IMS difieren en el "pegado" de la secuencia de segmentos dentro de un registro de base de datos física y en el tipo de estructura de acceso que se proporciona para localizar el registro de base de datos física o las ocurrencias de segmento individuales dentro de él.
- A partir del tipo de acceso a los registros de base de datos física provisto, IMS ofrece dos tipos de estructuras: **jerárquica secuencial** (HS: *hierarchical sequential*) y **jerárquica directa** (HD: *hierarchical direct*). Éstas se subdividen en HSAM, HISAM, HDAM y HIDAM, como se muestra en la figura 11.19.

Los métodos de acceso de IMS se pueden considerar como de alto nivel. IMS suministra rutinas para HSAM, HDAM, etc., que a su vez usan los métodos de acceso de bajo nivel llamados SAM (*sequential access method*: método de acceso secuencial), OSAM (*overflow sequential access method*: método de acceso secuencial con desborde) e ISAM (*indexed sequential access method*: método de acceso secuencial indizado). El método de acceso de alto nivel usa una combinación de archivos en el método de acceso de bajo nivel (véase la Fig. 11.19).

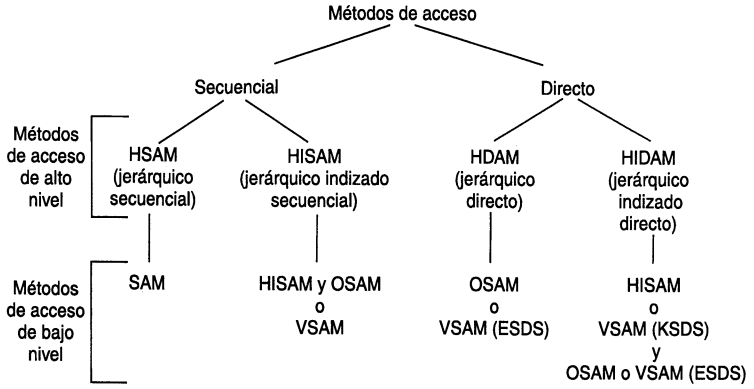


Figura 11.19 Panorama de los métodos de acceso de IMS.

HSAM. HSAM "pega" o "junta" los segmentos dentro de un árbol por *contigüidad física*. Los árboles en sí se colocan en secuencia física en el almacenamiento. Cada registro de base de datos física representa un árbol de ocurrencia. HSAM "encadena" los registros físicos secuencialmente, en orden de la clave de secuencia del segmento raíz, con tamaño de bloque fijo. Esta organización se parece a una cinta y sólo tiene importancia teórica, ya que no permite actualizaciones. Sin embargo, puede servir para vaciar y transportar bases de datos. Una vez cargada una base de datos (empleando órdenes ISRT), sólo se permiten las operaciones GET (con excepción de GET HOLD). Si es necesario modificar, insertar o eliminar datos, se lee la base de datos vieja y luego se escribe una copia nueva completa. Esta organización sirve para procesar datos que no cambian durante periodos largos.

HISAM. En la organización HISAM la base de datos consta de dos archivos o áreas de almacenamiento o conjuntos de datos: un archivo (el área principal) contiene segmentos raíz (más ciertos segmentos adicionales que caben dentro del registro en ese archivo); otro archivo (el área de desborde) contiene la porción restante de cada árbol linealizado. La figura 11.20 muestra cómo se almacenaría en IMS el registro de base de datos física correspondiente a la figura 11.7 usando HISAM. Puede verse cómo el registro se divide en dos archivos y cómo los bloques de longitud fija se enlazan dentro del desborde.

Ambos archivos tienen registros de longitud fija. En consecuencia, debido a las longitudes no uniformes de los segmentos, algo de espacio se desperdicia al final de los registros. El primer archivo es un área ISAM o bien VSAM accesible a través de un índice sobre el campo de secuencia del segmento raíz. Éste es un método de acceso muy común en IMS.

HDAM. Una base de datos HDAM consiste en un solo archivo OSAM o VSAM. En ambas estructuras HD, se tiene acceso al segmento raíz mediante una dispersión por campo de secuencia; los segmentos se almacenan de manera independiente y se enlazan mediante dos tipos de apuntadores:

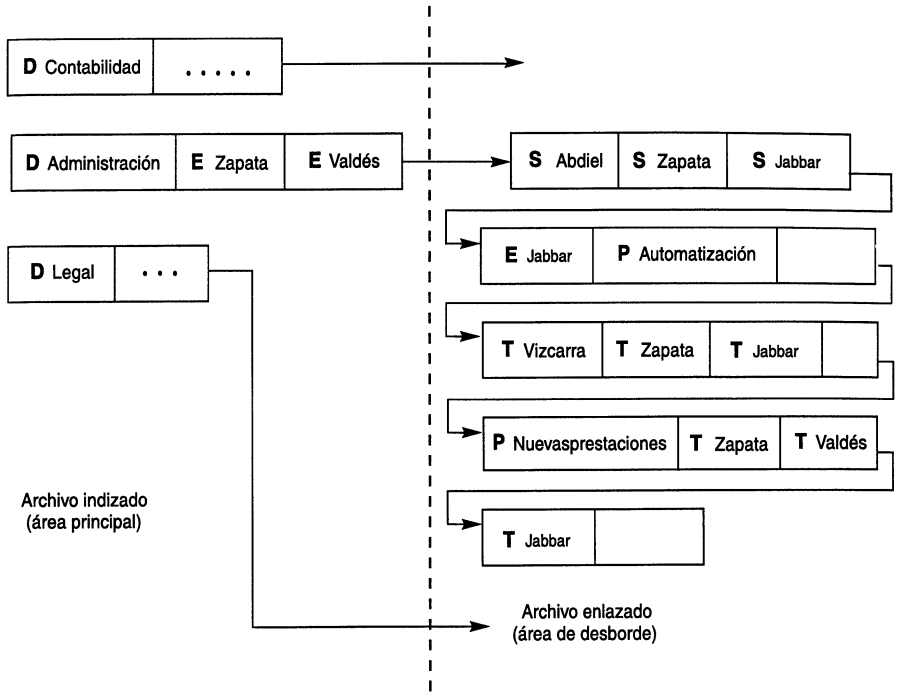


Figura 11.20 Organización de archivos HISAM en IMS.

- *Apuntadores jerárquicos:* Con los apuntadores jerárquicos, cada segmento apunta al siguiente en la secuencia jerárquica, excepto por el último segmento dependiente de la jerarquía, que no lleva apuntador. En esencia, éste es el recorrido en preorden del árbol.
- *Apuntadores hijo/gemelo:* La figura 11.21 muestra el empleo de los apuntadores hijo/gemelo. Cada tipo de segmentos tiene un *número designado* de apuntadores a hijos igual al número de tipos de segmentos hijo indicados en la definición de la base de datos, y *un* apuntador a gemelo. Para un árbol dado, el apuntador a hijo de un segmento puede ser (a) un valor nulo si no tiene un segmento hijo del tipo correspondiente (por ejemplo, ningún DEPENDIENTE para Zapata o Jabbar en la figura 11.21) o (b) la ubicación del primer segmento hijo del tipo correspondiente. Los hijos del mismo padre se enlazan con un apuntador a gemelo. El apuntador a gemelo del último hijo es nulo.

Cabe señalar que la organización HDAM *no ofrece acceso secuencial por el segmento raíz*, pero es posible obtenerlo mediante un índice secundario. También es posible crear apuntadores hacia atrás además de los apuntadores hacia adelante. La declaración de apuntadores se efectúa como parte de la definición de la base de datos.

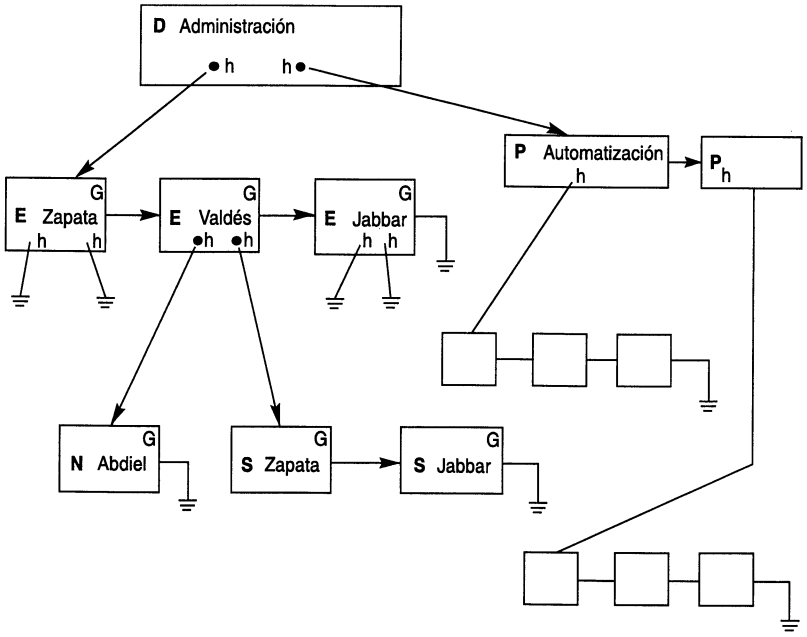


Figura 11.21 HDAM con apuntadores hijo/gemelo en IMS.

Desde el punto de vista del rendimiento, los apuntadores jerárquicos son preferibles cuando se requiere acceso directo a la raíz y acceso secuencial por segmentos dependientes, como en la producción de informes, donde deben listarse varios segmentos diferentes. Los apuntadores hijo/gemelo son preferibles cuando se desea acceso rápido a los niveles inferiores de la jerarquía o a la parte inferior derecha del árbol. HDAM difiere de los otros tres métodos de acceso en que la carga inicial de la base de datos puede hacerse en cualquier orden (aleatorio), árbol por árbol.

HIDAM. Una base de datos HIDAM consiste en dos partes: una base de datos índice y una base de datos de "datos". Esta última es un solo archivo OSAM o VSAM (conjunto de datos en orden de entrada) que consta de registros de longitud fija inicialmente cargados en secuencia jerárquica. Se le trata como una base de datos HDAM por sí sola, y para enlazar los segmentos se emplean esquemas de apuntadores jerárquicos o bien hijo/gemelo.

La base de datos índice es una base de datos HISAM en la que la jerarquía consta de un solo tipo de segmento, un segmento índice. Cada uno de éstos contiene un valor de campo de secuencia raíz como clave y el apuntador a ese segmento raíz en la base de datos como campo de datos. La base de datos índice es mucho más pequeña que la de datos. Si se usa VSAM, un conjunto de datos en secuencia por clave basta como base de datos de índice. El aprovechamiento del espacio por parte de la base de datos índice es más eficiente con VSAM que con ISAM/OSAM. Las bases de datos HIDAM se usan mucho porque combinan las ventajas de HISAM y de HDAM. El acceso directo a los segmentos raíz de la base de datos se logra usando la clave de secuencia raíz como clave de dispersión; el acceso indizado está disponible a través de la base de datos de índice.

Otras estructuras de almacenamiento IMS. Además de los métodos de acceso antes descritos, IMS cuenta con las siguientes estructuras de almacenamiento adicionales:

1. **HSAM simple** (SHSAM) **y HISAM simple** (SHISAM) son variantes de HSAM y HISAM, respectivamente, en las que la base de datos contiene sólo un tipo de segmentos (el segmento raíz).
2. **La característica de camino rápido** de IMS está diseñada para sistemas de transacciones en línea con altas tasas de transacciones y procesamiento relativamente simple. Ofrece recursos de comunicación de datos y dos estructuras especiales de base de datos:
   a. **Bases de datos de memoria principal** (MSDB: *main storage databases*): Una MSDB es una base de datos exclusivamente de raíz. Se mantiene en la memoria principal durante toda la operación del sistema. Las tablas de referencia pequeñas, como las tablas de conversión y los horarios, son buenos candidatos para MSDB.
   b. **Bases de datos de introducción de datos** (DEDB: *data entry databases*): Una DEDB es una forma especial de base de datos HDAM diseñada para mejorar la disponibilidad y el rendimiento. Es una forma restringida de jerarquía con sólo dos niveles, y puede estar dividida en hasta 240 áreas. El segmento del extremo izquierdo del segundo nivel, llamado *tipo de segmentos dependiente secuencial*, recibe un tratamiento especial. Cada área es un conjunto de datos VSAM individual, y cada registro de la base de datos (raíz más todos los dependientes) está contenido totalmente en el área. Las particiones son invisibles para la aplicación.
3. **Grupos de conjuntos de datos** (DSG: *data set groups*) **secundarios**: Una base de datos HISAM, HDAM o HIDAM puede dividirse en grupos de tipos de segmentos. Es posible crear un grupo de conjunto de datos primario y nueve grupos de conjuntos de datos secundarios. El DSG primario contiene el segmento raíz. Cada DSG secundario es una base de datos individual que contiene todas las ocurrencias de segmentos del tipo de segmentos que pertenece a él.

Indización secundaria en IMS. En el capítulo 5 vimos la importancia de los índices secundarios para reducir los tiempos de acceso en diversos tipos de archivos. IMS contempla sólo dos tipos de índices secundarios, a saber:

1. Un índice que provee acceso a un segmento raíz o dependiente con base en el valor de cualquiera de sus campos.
2. Un índice que indiza un segmento dado con base en un campo de algún *segmento de un nivel inferior*. En la base de datos de la figura 11.5, algunos de los posibles índices secundarios son:
   A. Un índice de DEPARTAMENTO por nombre de departamento (NOMBRED).
   B. Un índice de DEPENDIENTE por fecha de nacimiento (FECHANAC).
   C. Un índice de DEPARTAMENTO por ubicación de un proyecto de ese departamento (LUGARP en PROYECTO).

El procesamiento de índices secundarios en IMS adolece de dos defectos:

1. El código en DL/1 debe referirse explícitamente a un índice para poder usarlo; de lo contrario, el procesamiento se efectúa *sin* el índice.

2. Cuando se usa un campo de un segmento de nivel inferior en la jerarquía para la indización, la jerarquía se visualiza como si estuviera reestructurada con ese segmento como raíz.

Ambas características violan directamente el objetivo de independencia con respecto a los datos (véase la Sec. 2.2), por la cual la vista externa de un usuario queda aislada de la organización interna de la base de datos.

11.8 Resumen

En este capítulo estudiamos el modelo jerárquico, que representa los datos haciendo hincapié en los vínculos jerárquicos. La exposición tocó el tema de manera general, aunque en algunos aspectos seguimos el patrón del principal sistema jerárquico: IMS de IBM. En cuanto a las diferencias con respecto a IMS, casi todas las destacamos (aunque sin tratarlas a fondo) en el texto y en las notas a pie de página. Las principales estructuras que usa el modelo jerárquico son los tipos de registros y los tipos de vínculos padre-hijo (VPH). Cada tipo de VPH define un vínculo jerárquico 1:*N* entre un tipo de registros padre y un tipo de registros hijo. Los vínculos son estrictamente jerárquicos en el sentido de que un tipo de registros puede participar como hijo en, como máximo, un tipo de VPH. Esta restricción dificulta la representación de una base de datos en la que haya muchos vínculos.

En seguida vimos cómo se pueden definir los esquemas de bases de datos jerárquicas en forma de múltiples esquemas jerárquicos de tipos de registros. Un esquema jerárquico es básicamente una estructura de datos de árbol. En correspondencia con un esquema jerárquico, en la base de datos habrá varios árboles de ocurrencia. La secuencia jerárquica para almacenar registros de base de datos a partir de un árbol de ocurrencia es un recorrido en preorden de los registros de dicho árbol. El tipo de cada registro se almacena junto con el registro para que el SGBD pueda identificar los registros al examinarlos en el orden de la secuencia jerárquica.

Después examinamos las limitaciones que tiene la representación jerárquica cuando se intenta representar vínculos M:*N* o vínculos en los que participen más de dos tipos de registros. Es posible representar algunos de estos casos si permitimos la existencia de registros redundantes en la base de datos. Usamos el concepto de vínculo padre-hijo virtual (VPHV) para que un tipo de registros pueda tener dos padres: uno real y uno virtual. Este tipo de VPHV también puede servir para representar vínculos M:*N* sin redundancia de registros. También tratamos los tipos de restricciones de integridad implícitas en las jerarquías.

En la sección 11.5 estudiamos el diseño de bases de datos jerárquicas a partir de un esquema conceptual ER. En general, el modelo jerárquico funciona bien con aplicaciones de bases de datos que sean naturalmente jerárquicas. Sin embargo, cuando hay muchos vínculos no jerárquicos, es difícil tratar de ajustar dichos vínculos a una forma jerárquica, y a menudo las representaciones resultantes son insatisfactorias. A continuación presentamos las órdenes de un lenguaje de definición de datos jerárquicos hipotético (HDDL) y de un lenguaje de manipulación de datos jerárquicos de registro por registro (HDML). El HDML se basa en la secuencia jerárquica. Vimos la manera de escribir programas con órdenes de HDML incorporadas para obtener información de una base de datos jerárquica y para actualizar dicha base de datos.

Por último, presentamos la arquitectura básica, las características de lenguaje y la organización de almacenamiento de un sistema de bases de datos jerárquicas muy utilizado, IMS.

Aunque el modelo relacional y los SGBD relacionales se han popularizado mucho a últimas fechas, el modelo jerárquico se seguirá utilizando durante varios años más debido a la enorme inversión que se ha destinado a los SGBD jerárquicos en el mundo comercial. Además, el modelo jerárquico es idóneo para situaciones en las que la mayoría de los vínculos son jerárquicos y en las que el acceso a las bases de datos aprovecha principalmente estos vínculos.

Preguntas de repaso

11.1. Defina los siguientes términos: *tipo de vínculos padre-hijo (VPH), raíz de una jerarquía, hoja de una jerarquía.*

11.2. Analice las principales propiedades de una jerarquía.

11.3. Explique los problemas que arroja el empleo de un tipo de VPH para representar un vínculo M:*N*.

11.4. ¿Qué es un árbol de ocurrencia de una jerarquía?

11.5. ¿Qué es la secuencia jerárquica? ¿Por qué es necesario asignar un campo de tipo de registros a cada registro cuando se usa la secuencia jerárquica para representar un árbol de ocurrencia?

11.6. Defina los siguientes términos: *camino jerárquico, retama, bosque de árboles, base de datos jerárquica.*

11.7. ¿Qué son los tipos de vínculos padre-hijo virtuales (VPHV)? ¿En qué sentido aumentan la capacidad de modelado del modelo jerárquico?

11.8. Analice las diferentes técnicas con que se pueden implementar los tipos de VPHV en una base de datos jerárquica.

11.9. Analice las restricciones de integridad inherentes del modelo jerárquico.

11.10. Indique cómo se representan los siguientes tipos de vínculos en el modelo jerárquico: (a) vínculos M:*N*; (b) vínculos *n*-arios, con $n > 2$; (c) vínculos 1:1. Explique la forma de transformar un esquema ER a un esquema jerárquico.

11.11. ¿Por qué es necesario incorporar las órdenes de HDML en un lenguaje de programación anfitrión como PASCAL?

11.12. Explique los siguientes conceptos, e identifique la utilidad de cada uno cuando se escribe un programa de base de datos en HDML: (a) área de trabajo del usuario (UWA); (b) indicadores de actualidad; (c) indicador de estado de la base de datos.

11.13. Analice los diferentes tipos de órdenes GET del HDML, indicando cómo afecta cada uno los indicadores de actualidad.

11.14. Explique cómo se establecen los registros padre como resultado de una orden de obtención de datos. ¿Por qué la orden GET NEXT WITHIN PARENT *no* establece un nuevo padre?

11.15. Analice las órdenes de actualización de HDML.

Ejercicios

11.16. Especifique las consultas del ejercicio 6.19 en HDML incorporado en PASCAL sobre el esquema de base de datos jerárquica de la figura 11.10. Use las variables de programa PASCAL declaradas en la figura 11.11, y declare cualesquier variables adicionales que necesite.

11.17. Considere el esquema de base de datos jerárquica ilustrado en la figura 11.22, que corresponde al esquema relacional de la figura 2.1. Escriba enunciados apropiados en HDDL para definir los tipos de registros del esquema.

11.18. El esquema de la figura 11.22 contiene cierta redundancia; ¿cuáles elementos de información se repiten de manera redundante? ¿Puede especificar un esquema de base de datos jerárquica para esta base de datos sin redundancia usando VPHV?

11.19. Escriba segmentos de programa en PASCAL con órdenes de HDML incorporadas para especificar las consultas del ejercicio 7.16 sobre el esquema de la figura 11.22. Repita las mismas consultas sobre el esquema que haya preparado para el ejercicio 11.18.

11.20. Escriba segmentos de programa en PASCAL con órdenes de HDML incorporadas para efectuar las actualizaciones y tareas de los ejercicios 7.17 y 7.18 sobre el esquema de base de datos jerárquica de la figura 11.22. Especifique cualesquier variables de programa que necesite. Repita las mismas consultas sobre su esquema del ejercicio 11.18.

11.21. Escoja alguna aplicación de base de datos que conozca o le interese.

a. Diseñe un esquema de base de datos jerárquica para esa aplicación.

b. Declare sus tipos de registros, tipos de VPH y tipos de VPHV, usando el HDDL.

c. Especifique varias consultas y actualizaciones que necesite su aplicación de base de datos y escriba un segmento de programa en PASCAL con órdenes de HDML incorporadas para cada una de sus consultas.

d. Implemente su base de datos si tiene acceso a un SGBD jerárquico.

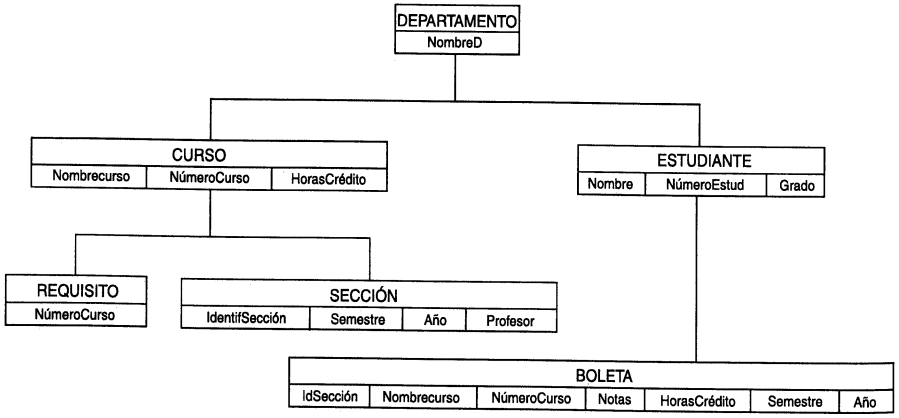


Figura 11.22 Esquema jerárquico para una base de datos universitaria.

11.22. Establezca la transformación de los siguientes esquemas ER a esquemas jerárquicos. Para cada esquema ER, especifique uno o más esquemas jerárquicos, indicando cuáles tienen redundancias y cuáles no.

a. El esquema ER AEROLÍNEA de la figura 3.19.

b. El esquema ER BANCO de la figura 3.20.

c. El esquema ER CONTROL_BUQUES de la figura 6.21.

Bibliografía selecta

IBM y North American Aviation (Rockwell International) desarrollaron el primer SGBD jerárquico —IMS y su lenguaje DL/1— a finales de la década de 1960. Son pocos los documentos de esa época que describan IMS. McGee (1977) presenta un panorama sobre IMS en un número de IBM *Systems Journal* dedicado a IMS. Bjoerner y Lovengren (1982) formalizan algunos aspectos del modelo de datos de IMS.

El Time-shared Data Management System (TDMS: sistema de gestión de datos de tiempo compartido), de System Development Corporation (ahora Burroughs) (Vorhaus y Mills 1967; Bleier y Vorhaus 1968), y el Remote File Management System (RFMS: sistema remoto de gestión de archivos), creado en la University of Texas en Austin (Everett *et al.* 1971), son precursores de otro importante sistema jerárquico que se encuentra en el mercado, el System 2000, y que ahora comercializa SAS Inc. Hardgrave (1974, 1980) describe un lenguaje, BOLT, para el modelo jerárquico.

Tsichritzis y Lochovsky (1976) estudian la gestión de bases de datos jerárquicas, y de entre varios libros de texto en que se hacen descripciones generales sobre el modelo jerárquico podemos destacar Korth y Silberschatz (1991). Kroenke y Dolan (1988) analiza el procesamiento con DL/1, y Date (1990) presenta IMS como ejemplo de sistema jerárquico. Kapp y Leben (1978) analiza en detalle el lenguaje DL/1 desde el punto de vista del programador. Trabajos recientes han intentado incorporar estructuras jerárquicas en el modelo relacional (Gyssens *et al.* 1989; Jagadish 1989); en ellos se tratan los modelos relacionales anidados (véase la Sec. 21.6.2).

CAPÍTULO 12

Dependencias funcionales y normalización para bases de datos relacionales

En los capítulos 6 a 8 presentamos varios aspectos del modelo relacional, con algunos ejemplos de bases de datos relacionales. Cada *esquema de relación* consiste en diversos atributos, y el *esquema de base de datos relacional* consta de algunos esquemas de relación. Hasta aquí, hemos supuesto que los atributos se agrupan para formar un esquema de relación empleando el sentido común del diseñador de bases de datos o estableciendo una transformación de un esquema especificado en el modelo de entidad-vínculo a un esquema relacional. Sin embargo, no hemos contado con una medida formal de por qué una agrupación de atributos para formar un esquema de relación puede ser mejor que otra. No nos fundamos en ninguna medida de lo apropiado del diseño o de su calidad que no fuera la mera intuición del diseñador.

En este capítulo estudiaremos parte de la teoría que se ha desarrollado en un intento por elegir "buenos" esquemas de relación; esto es, por medir formalmente las razones por las que una agrupación de atributos en esquemas de relación es mejor que otra. Hay dos niveles en los que podemos evaluar la "bondad" de los esquemas de relación. El primero es el **nivel lógico**, que se refiere a la manera en que los usuarios interpretan los esquemas de relación y el significado de sus atributos. Contar con buenos esquemas de relación en este nivel ayuda a los usuarios a comprender con claridad el significado de las tuplas de datos en las relaciones, y por tanto a formular sus consultas correctamente. El segundo es el **nivel de manipulación** (o de **almacenamiento**), que se refiere a cómo se almacenan y actualizan las tuplas de una relación base. Este nivel sólo se aplica a los esquemas de relaciones base —que se almacenarán físicamente como archivos— mientras que en el nivel lógico nos interesan los esquemas tanto de las relaciones base como de las vistas (relaciones virtuales). La teoría para el diseño de bases de datos relacionales expuesta en este capítulo vale sobre todo para las relaciones base, aunque algunos criterios de idoneidad se aplican también a las vistas, como veremos en la sección 12.1.

Comenzamos en la sección 12.1 con un análisis informal de algunos criterios para distinguir los esquemas de relación buenos y malos. Luego, en la sección 12.2, definiremos el concepto de dependencia funcional, que es la principal herramienta para medir formalmente la idoneidad de las agrupaciones de atributos en los esquemas de relación. También estudiaremos y analizaremos las propiedades de las dependencias funcionales. En la sección 12.3 mostraremos cómo usar las dependencias funcionales para agrupar atributos en esquemas de relación que estén en una forma normal. Un esquema de relación está en una forma normal cuando posee ciertas características deseables. Es posible definir formas normales para los esquemas de relación que dan lugar a agrupaciones progresivamente mejores. En la sección 12.4 estableceremos definiciones más generales de algunas formas normales.

El capítulo 13 continúa el desarrollo de la teoría relacionada con el diseño de buenos esquemas relacionales. Mientras que en el capítulo 12 nos concentraremos en las formas normales para esquemas de una sola relación, en el capítulo 13 estudiaremos las medidas de idoneidad para cualquier conjunto de esquemas de relación que globalmente constituya un esquema de base de datos relacional. Especificaremos dos de esas propiedades —la propiedad de reunión no aditiva (sin pérdida) y la propiedad de conservación de la dependencia— y analizaremos algoritmos para diseñar bases de datos relacionales a partir de las dependencias funcionales, las formas normales y las propiedades que acabamos de mencionar. En el capítulo 13 definiremos también otros tipos de dependencias y formas normales avanzadas que mejoran aún más las propiedades de los esquemas de relación.

El lector que sólo busque una explicación informal sobre la normalización puede pasar por alto las secciones 12.1.4, 12.2.3, 12.2.4 y 12.5.

12.1 Pautas informales de diseño para los esquemas de relaciones

En esta sección analizaremos cuatro *medidas informales* de la calidad para el diseño de esquemas de relaciones:

- Semántica de los atributos.
- Reducción de los valores redundantes en las tuplas.
- Reducción de los valores nulos en las tuplas.
- Prohibición de tuplas espurias.

Estas medidas no siempre son independientes entre sí, como veremos.

12.1.1 *Semántica de los atributos de una relación*

Siempre que agrupemos atributos para formar un esquema de relación, supondremos que hay un cierto significado asociado a los atributos. En el capítulo 6 vimos cómo cada relación puede interpretarse como un conjunto de hechos o enunciados. Este significado, o **semántica**, especifica cómo se han de interpretar los valores de los atributos almacenados en una tupla de la relación; dicho de otro modo, qué relación hay entre los valores de los atributos de una tupla. Cuanto más fácil sea explicar la semántica de la relación, mejor será el diseño del esquema correspondiente.

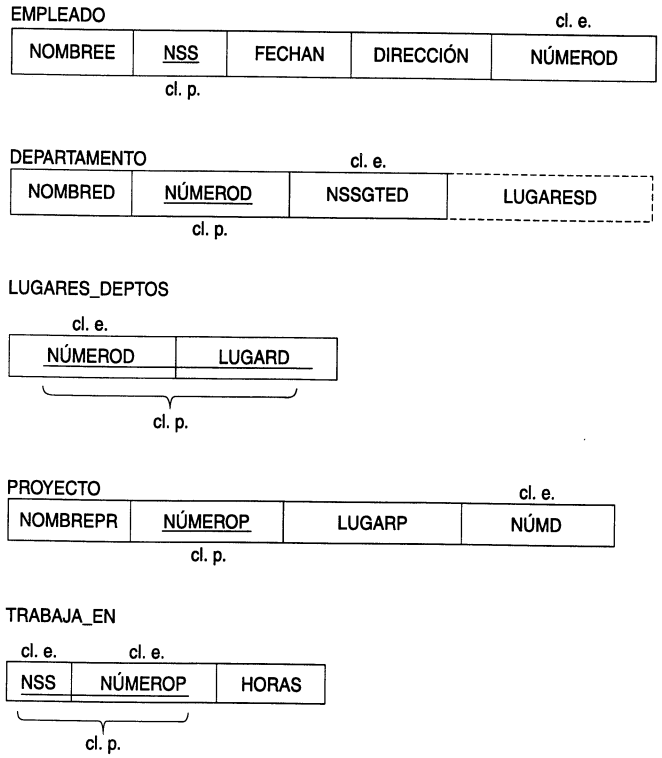


Figura 12.1 Versión simplificada del esquema de la base de datos relacional COMPAÑÍA.

Para ilustrar esto, consideremos una versión simplificada del esquema de base de datos relacional COMPAÑÍA de la figura 6.5, que aparece en la figura 12.1. En la figura 12.2 se muestran relaciones de ejemplo para este esquema. El significado del esquema de relación EMPLEADO es muy sencillo: cada tupla representa un empleado, con valores para su nombre (NOMBREE), número de seguro social (NSS), fecha de nacimiento (FECHAN) y domicilio (DIRECCIÓN), y para el número del departamento al que pertenece (NÚMEROD). El atributo NÚMEROD es una clave externa que representa un *vínculo implícito* entre EMPLEADO y DEPARTAMENTO. La semántica de los esquemas DEPARTAMENTO y PROYECTO también es evidente; cada tupla DEPARTAMENTO representa una entidad departamento, y cada tupla PROYECTO representa una entidad proyecto. El atributo NSSGTED de DEPARTAMENTO relaciona un departamento con el empleado que es su gerente, y NÚMD de PROYECTO relaciona un proyecto con el departamento que lo controla; ambos son atributos de clave externa. Por el momento, el lector debe hacer caso omiso del atributo LUGARESD de DEPARTAMENTO, pues se usará para ilustrar los conceptos de normalización en la sección 12.3.

La semántica de los otros dos esquemas de relación de la figura 12.1 es un poco más compleja. Cada tupla de LUGARES_DEPTOS contiene un número de departamento (NÚMEROD)

EMPLEADO

NOMBREE	NSS	FECHAN	DIRECCIÓN	NÚMEROD
Silva, José B.	123456789	09-ENE-55	Fresnos 731, Higueras, MX	5
Vizcarra, Federico T.	333445555	08-DIC-45	Valle 638, Higueras, MX	5
Zapata, Alicia J.	999887777	19-JUL-58	Castillo 3321, Sucre, MX	4
Valdés, Jazmín S.	987654321	20-JUN-31	Bravo 291, Belén, MX	4
Nieto, Ramón K.	666884444	15-SEP-52	Espiga 957, Heras, MX	5
Esparza, Josefa A.	453453453	31-JUL-62	Rosas 5631, Higueras, MX	5
Jabbar, Ahmed V.	987987987	29-MAR-59	Dalias 980, Higueras, MX	4
Botello, Jaime E.	888665555	10-NOV-27	Sorgo 450, Higueras, MX	1

DEPARTAMENTO

NOMBRED	NÚMEROD	NSSGTED
Investigación	5	333445555
Administración	4	987654321
Dirección	1	888665555

LUGARES_DEPTOS

NÚMEROD	LUGARD
1	Higueras
4	Santiago
5	Belén
5	Sacramento
5	Higueras

TRABAJA_EN

NSS	NÚMEROP	HORAS
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	nulo

PROYECTO

NOMBREP	NÚMEROP	LUGARP	NÚMD
ProductoX	1	Belén	5
ProductoY	2	Sacramento	5
ProductoZ	3	Higueras	5
Automatización	10	Santiago	4
Reorganización	20	Higueras	1
Nuevasprestaciones	30	Santiago	4

Figura 12.2 Ejemplos de relaciones para el esquema de la figura 12.1.

y *una de* las ubicaciones del departamento (LUGARD). Cada tupla de TRABAJA_EN contiene el número de seguro social de un empleado (NSS), el número de *uno de* los proyectos en los que ese empleado trabaja (NÚMEROP) y el número de horas por semana que el empleado trabaja en el proyecto (HORAS). No obstante, ambos esquemas tienen un significado bien definido, sin ambigüedad. El esquema LUGARES_DEPTOS representa un atributo multivaluado de DEPARTAMENTO, en tanto que TRABAJA_EN representa un vínculo M:*N* entre EMPLEADO y PROYECTO. Así pues, todos los esquemas de relación de la figura 12.1 pueden considerarse buenos en lo tocante a la claridad de su semántica. Podemos enunciar la siguiente pauta para el diseño de un esquema de relación:

PAUTA 1: Diseñe un esquema de relación de modo que sea fácil explicar su significado. No combine atributos de varios tipos de entidades y tipos de vínculos en una sola relación. Intuitivamente, si un esquema de relación corresponde a un tipo de entidades o a un tipo de vínculos, el significado tiende a ser claro. En caso contrario, tiende a ser una mezcla de múltiples entidades y vínculos y, por tanto, será semánticamente confuso. ■

Los esquemas de relación de las figuras 12.3(a) y (b) también tienen semántica clara. (Por ahora, el lector deberá hacer caso omiso de las líneas que están abajo de las relaciones, pues servirán para ilustrar la notación de dependencias funcionales en la sección 12.2.) Una tupla del esquema de relación EMP_DEPTO de la figura 12.3(a) representa un solo empleado pero contiene información adicional; a saber, el nombre (NOMBRED) del departamento al que pertenece el empleado y el número de seguro social (NSSGTED) del gerente del departamento. En el caso de la relación EMP_PROY de la figura 12.3(b), cada tupla relaciona un empleado con un proyecto pero también incluye el nombre del empleado (NOMBREE), el nombre del proyecto (NOMBREPR) y la ubicación del proyecto (LUGARP). Aunque no hay nada incorrecto en estas relaciones desde el punto de vista lógico, se les considera diseños deficientes porque violan la pauta 1 al *mezclar atributos de entidades distintas del mundo real*; EMP_DEPTO mezcla atributos de empleados y departamentos y EMP_PROY mezcla atributos de empleados y proyectos. Pueden servir como vistas, pero provocan problemas cuando se usan como relaciones base, como veremos en la siguiente sección.

12.1.2 *Información redundante en las tuplas y anomalías de actualización*

Uno de los objetivos en el diseño de esquemas es minimizar el espacio de almacenamiento que ocupan las relaciones base (archivos). La agrupación de atributos en esquemas de relación tiene un efecto significativo sobre el espacio de almacenamiento. Por ejemplo, basta con comparar el espacio utilizado por las dos relaciones base EMPLEADO y DEPARTAMENTO de la figura 12.2 con el espacio de la relación base EMP_DEPTO de la figura 12.4, que es el resultado de aplicar la operación de reunión natural a EMPLEADO y DEPARTAMENTO. En

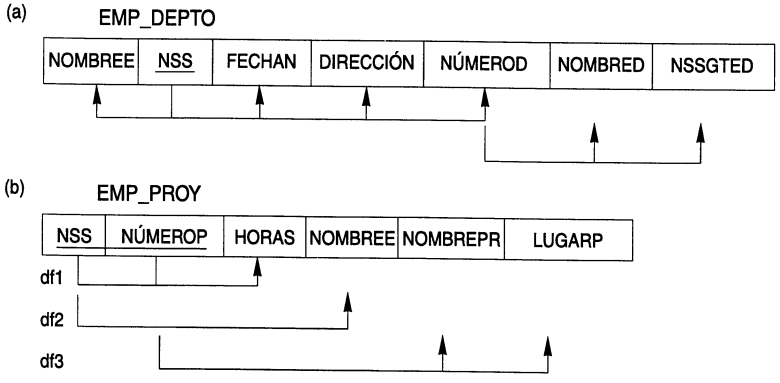


Figura 12.3 Dos esquemas de relación y sus dependencias funcionales. (a) El esquema de relación EMP_DEPTO. (b) El esquema de relación EMP_PROY.

EMP_DEPTO, los valores de los atributos pertenecientes a un departamento en particular (NÚMEROD, NOMBRED, NSSGTED) se repiten para cada empleado que trabaja para ese departamento. En cambio, la información de cada departamento aparece sólo una vez en la relación DEPARTAMENTO de la figura 12.2. Sólo el número del departamento (NÚMEROD) se repite en la relación EMPLEADO para cada empleado que pertenece al departamento. Podemos hacer comentarios similares respecto a la relación EMP_PROY (Fig. 12.4) que hace crecer la relación TRABAJA_EN con atributos adicionales de EMPLEADO y PROYECTO.

Otro problema grave que surge al emplear las relaciones de la figura 12.4 como relaciones base es el de las **anomalías de actualización.** Éstas pueden clasificarse en anomalías de inserción, anomalías de eliminación y anomalías de modificación.†

Anomalías de inserción. Éstas pueden diferenciarse en dos tipos, que ilustramos con los siguientes ejemplos basados en la relación EMP_DEPTO:

- Si queremos insertar la tupla de un nuevo empleado en EMP_DEPTO, debemos incluir los valores de los atributos del departamento al que pertenece, o bien nulos (si

EMP_DEPTO

NOMBREE	NSS	FECHAN	DIRECCIÓN	NÚMEROD	NOMBRED	NSSGTED
Silva, José B.	123456789	09-ENE-55	Fresnos 731, Higueras, MX	5	Investigación	333445555
Vizcarra, Federico T.	333445555	08-DIC-45	Valle 638, Higueras, MX	5	Investigación	333445555
Zapata, Alicia J.	999887777	19-JUL-58	Castillo 3321, Sucre, MX	4	Administración	987654321
Valdés, Jazmín S.	987654321	20-JUN-31	Bravo 291, Belén, MX	4	Administración	987654321
Nieto, Ramón K.	666884444	15-SEP-52	Espiga 957, Heras, MX	5	Investigación	333445555
Esparza, Josefa A.	453453453	31-JUL-62	Rosas 5631, Higueras, MX	5	Investigación	333445555
Jabbar, Ahmed V.	987987987	29-MAR-59	Dalias 980, Higueras, MX	4	Administración	987654321
Botello, Jaime E.	888665555	10-NOV-27	Sorgo 450, Higueras, MX	1	Dirección	888665555

EMP_PROY

NSS	NÚMEROP	HORAS	NOMBREE	NOMBREPR	LUGARP
123456789	1	32.5	Silva, José B.	ProductoX	Belén
123456789	2	7.5	Silva, José B.	ProductoY	Sacramento
666884444	3	40.0	Nieto, Ramón K.	ProductoZ	Higueras
453453453	1	20.0	Esparza, Josefa A.	ProductoX	Sacramento
453453453	2	20.0	Esparza, Josefa A.	ProductoY	Sacramento
333445555	2	10.0	Vizcarra, Federico T.	ProductoY	Sacramento
333445555	3	10.0	Vizcarra, Federico T.	ProductoZ	Higueras
333445555	10	10.0	Vizcarra, Federico T.	Automatización	Santiago
333445555	20	10.0	Vizcarra, Federico T.	Reorganización	Higueras
999887777	30	30.0	Zapata, Alicia J.	Nuevasprestaciones	Santiago
999887777	10	10.0	Zapata, Alicia J.	Automatización	Santiago
987987987	10	35.0	Jabbar, Ahmed V.	Automatización	Santiago
987987987	30	5.0	Jabbar, Ahmed V.	Nuevasprestaciones	Santiago
987654321	30	20.0	Valdés, Jazmín S.	Nuevasprestaciones	Santiago
987654321	20	15.0	Valdés, Jazmín S.	Reorganización	Higueras
888665555	20	nulo	Botello, Jaime E.	Reorganización	Higueras

Figura 12.4 Ejemplos de relaciones para los esquemas de la figura 12.3 que resultan de aplicar una REUNIÓN NATURAL a las relaciones de la figura 12.2.

†Codd (1972*a*) identificó estas anomalías para justificar la necesidad de normalizar estas relaciones, como veremos en la sección 12.3.

el empleado todavía no pertenece a ningún departamento). Por ejemplo, para insertar una nueva tupla de un empleado que pertenece al departamento 5, debemos introducir correctamente los valores del departamento 5 para que sean *congruentes* con los valores del departamento 5 en otras tuplas de EMP_DEPTO. En el diseño de la figura 12.2 no tenemos que preocuparnos por este problema de congruencia ya que sólo introduciremos el número del departamento en la tupla del empleado; todos los demás atributos del departamento 5 se registrarán una sola vez en la base de datos, como una tupla única de la relación DEPARTAMENTO.

- Es difícil insertar en la relación EMP_DEPTO un nuevo departamento que todavía no tenga empleados. La única forma de hacer esto es colocar valores nulos en los atributos de empleado. Esto originará problemas porque NSS es la clave primaria de EMP_DEPTO, y se supone que cada tupla representa una entidad empleado, no una entidad departamento. Es más, cuando se asigne el primer empleado a ese departamento ya no necesitaremos la tupla con valores nulos. Este problema no se presentará en el diseño de la figura 12.2, porque los departamentos se introducen en la relación DEPARTAMENTO sin importar si algún empleado pertenece a ellos o no; y siempre que un empleado se asigna a ese departamento se insertará una tupla correspondiente en EMPLEADO.

Anomalías de eliminación. Este problema se relaciona con la segunda situación de anomalía de inserción que acabamos de ver. Si eliminamos de EMP_DEPTO una tupla de empleado que representa al último empleado perteneciente a un cierto departamento, la información concerniente a ese departamento se perderá de la base de datos. Este problema no ocurre en la base de datos de la figura 12.2, porque las tuplas de DEPARTAMENTO se almacenan aparte.

Anomalías de modificación. En EMP_DEPTO, si alteramos el valor de uno de los atributos de un departamento dado —digamos, el gerente del departamento 5— deberemos actualizar las tuplas de todos los empleados que pertenezcan a ese departamento; de lo contrario, la base de datos se volverá inconsistente. Si dejamos algunas tuplas sin actualizar, el mismo departamento tendrá dos valores de gerente distintos en diferentes tuplas de empleado, lo que no debe suceder.

Con base en las tres anomalías anteriores, podemos enunciar la siguiente pauta:

PAUTA 2: Diseñe los esquemas de las relaciones base de modo que no haya anomalías de inserción, eliminación o modificación en las relaciones. Si hay anomalías, señálelas con claridad a fin de que los programas que actualicen la base de datos operen correctamente. ■

La segunda pauta es congruente con la primera y, en cierta forma, es otro modo de expresarla. También podemos percatarnos de la necesidad de un mecanismo más formal para evaluar si un diseño sigue o no estas pautas. En las secciones 12.2 a 12.4 se presentarán estos conceptos formales necesarios. Es importante señalar que hay ocasiones en que *es preciso violar* estas pautas para *mejorar el rendimiento* de ciertas consultas. Por ejemplo, si una consulta importante obtiene información relativa al departamento de un empleado, junto con los atributos del empleado, el esquema EMP_DEPTO puede servir como relación base. Sin embargo, debemos tomar nota de las anomalías de EMP_DEPTO y entenderlas

perfectamente de modo que, al actualizar la relación base, no se produzcan inconsistencias. En general, es aconsejable usar relaciones base libres de anomalías y especificar vistas que incluyan las reuniones para colocar juntos los atributos a los que frecuentemente se hace referencia en consultas importantes. Esto reduce el número de términos de reunión especificados en la consulta, lo que facilitará su escritura correcta y en muchos casos[†] mejorará el rendimiento.

12.1.3 *Valores nulos en las tuplas*

En algunos diseños de esquemas quizá agruparemos muchos atributos para formar una relación "gruesa". Si muchos de los atributos no se aplican a todas las tuplas de la relación, acabaremos con un gran número de nulos en esas tuplas. Esto puede originar un considerable desperdicio de espacio en el nivel de almacenamiento y posiblemente dificultar el entendimiento del significado de los atributos y la especificación de operaciones de REUNIÓN en el nivel lógico. Otro problema con los nulos es cómo manejarlos cuando se aplican funciones agregadas como COUNT o SUM. Además, los nulos pueden tener múltiples interpretaciones, como los siguientes:

- El atributo *no se aplica* a esta tupla.
- Se *desconoce* el valor del atributo para esta tupla.
- El valor *se conoce pero está ausente*; esto es, todavía no se ha registrado.

Tener la misma representación para todos los nulos puede hacer que se confundan los diferentes significados que podrían tener. Por tanto, podemos expresar otra pauta como sigue:

PAUTA 3: Hasta donde sea posible, evite incluir en una relación base atributos cuyos valores puedan ser nulos. Si no es posible evitar los nulos, asegúrese de que se apliquen sólo en casos excepcionales y no a la mayoría de las tuplas de una relación. ■

Por ejemplo, si sólo el 10% de los empleados tienen oficinas individuales, no se justificará incluir un atributo NÚM_OFICINA en la relación EMPLEADO; más bien, podríamos crear una relación OFICINAS_EMPS(NSSE, NÚM_OFICINA) que contenga exclusivamente tuplas para los empleados con oficinas individuales.

12.1.4 *Tuplas espurias*★

Consideremos los dos esquemas de relación LUGARES_EMPS y EMP_PROY1 de la figura 12.5(a), que podríamos usar en vez de la relación EMP_PROY de la figura 12.3(b). Una tupla en LUGARES_EMPS significa que el empleado cuyo nombre es NOMBREE trabaja en *algún proyecto* cuya ubicación es LUGARP. Una tupla en EMP_PROY1 significa que el empleado cuyo número de seguro social es NSS trabaja HORAS por semana en el proyecto cuyo nombre, número y ubicación son NOMBREPR, NÚMEROP y LUGARP. La figura 12.5(b) muestra extensiones de

[†]El rendimiento de una consulta especificada sobre una vista que es la reunión de varias relaciones base depende de la manera en que el SGBD implemente la vista. Muchos SGBD relacionales materializan una vista que se use con frecuencia para no tener que realizar las reuniones muy seguido. Sigue siendo obligación del SGBD actualizar automáticamente la vista materializada cada vez que se actualizan las relaciones base.

(a)

LUGARES_EMP

NOMBREE	LUGARP

c.l. p.

EMP_PROY1

NSS	NÚMEROP	HORAS	NOMBREPR	LUGARP

c.l. p.

(b)

LUGARES_EMP

NOMBREE	LUGARP
Silva, José B.	Belén
Silva, José B.	Sacramento
Nieto, Ramón K.	Higueras
Esparza, Josefa A.	Belén
Esparza, Josefa A.	Sacramento
Vizcarra, Federico T.	Sacramento
Vizcarra, Federico T.	Higueras
Vizcarra, Federico T.	Santiago
Zapata, Alicia J.	Santiago
Jabbar, Ahmed V.	Santiago
Valdés, Jazmín S.	Santiago
Valdés, Jazmín S.	Higueras
Botello, Jaime E.	Higueras

EMP_PROY1

NSS	NÚMEROP	HORAS	NOMBREPR	LUGARP
123456789	1	32.5	ProductoX	Belén
123456789	2	7.5	ProductoY	Sacramento
666884444	3	40.0	ProductoZ	Higueras
453453453	1	20.0	ProductoX	Belén
453453453	2	20.0	ProductoY	Sacramento
333445555	2	10.0	ProductoY	Sacramento
333445555	3	10.0	ProductoZ	Higueras
333445555	10	10.0	Automatización	Santiago
333445555	20	10.0	Reorganización	Higueras
999887777	30	30.0	Nuevasprestaciones	Santiago
999887777	10	10.0	Automatización	Santiago
987987987	10	35.0	Automatización	Santiago
987987987	30	5.0	Nuevasprestaciones	Santiago
987654321	30	20.0	Nuevasprestaciones	Santiago
987654321	20	15.0	Reorganización	Higueras
888665555	20	nulo	Reorganización	Higueras

Figura 12.5 Representación alternativa de EMP_PROY. (a) Representación de EMP_PROY de la figura 12.3(b) con dos esquemas de relación: LUGARES_EMPS y EMP_PROY1. (b) Resultado de proyectar la relación EMP_PROY de la figura 12.4 sobre los atributos de EMP_PROY1 y LUGARES_EMPS.

las relaciones LUGARES_EMPS y EMP_PROY1 que corresponden a la relación EMP_PROY de la figura 12.4 y que se obtienen aplicando las operaciones PROYECTAR (π) apropiadas a EMP_PROY (haga caso omiso por ahora de las líneas punteadas en la figura 12.5(b)).

Suponga que EMP_PROY1 y LUGARES_EMPS son las relaciones base, en vez de EMP_PROY. Esto produce un diseño de esquema particularmente insatisfactorio, porque no es posible recuperar la información contenida originalmente en EMP_PROY a partir de EMP_PROY1 y LUGARES_EMPS. Si intentamos una operación de REUNIÓN NATURAL con EMP_PROY1 y LUGARES_EMPS obtendremos muchas más tuplas de las que tenía EMP_PROY. En la figura 12.6 se muestra sólo el resultado de aplicar la reunión a las tuplas que están *arriba* de las líneas punteadas en la figura 12.5(b), para reducir el tamaño de la relación resultante. Las tuplas adicionales que no estaban en EMP_PROY se denominan **tuplas espurias** porque representan información espuria o *errónea* que no es válida. En la figura 12.6, las tuplas espurias están marcadas con asteriscos (*).

La descomposición de EMP_PROY en LUGARES_EMPS y EMP_PROY1 no es satisfactoria porque, cuando las reunimos otra vez con una REUNIÓN NATURAL, no obtenemos la información original correcta. Esto se debe a que LUGARP es el atributo que relaciona LUGARES_EMPS y EMP_PROY1, y LUGARP no es ni clave primaria ni clave externa en cualquiera de esas dos relaciones. Ahora podemos expresar informalmente otra pauta de diseño:

PAUTA 4: Diseñe los esquemas de relación de modo que puedan reunirse mediante condiciones de igualdad sobre atributos que sean claves primarias o claves externas, a fin de garantizar que no se formarán tuplas espurias. ■

	NSS	NÚMEROP	HORAS	NOMBREPR	LUGARP	NOMBREE
	123456789	1	32.5	ProductoX	Belén	Silva, José B.
*	123456789	1	32.5	ProductoX	Belén	Esparza, Josefa A.
	123456789	2	7.5	ProductoY	Sacramento	Silva, José B.
*	123456789	2	7.5	ProductoY	Sacramento	Esparza, Josefa A.
*	123456789	2	7.5	ProductoY	Sacramento	Vizcarra, Federico T.
	666884444	3	40.0	ProductoZ	Higueras	Nieto, Ramón K.
*	666884444	3	40.0	ProductoZ	Higueras	Vizcarra, Federico T.
*	453453453	1	20.0	ProductoX	Belén	Silva, José B.
	453453453	1	20.0	ProductoX	Belén	Esparza, Josefa A.
*	453453453	2	20.0	ProductoY	Sacramento	Silva, José B.
	453453453	2	20.0	ProductoY	Sacramento	Esparza, Josefa A.
*	453453453	2	20.0	ProductoY	Sacramento	Vizcarra, Federico T.
*	333445555	2	10.0	ProductoY	Sacramento	Silva, José B.
*	333445555	2	10.0	ProductoY	Sacramento	Esparza, Josefa A.
	333445555	2	10.0	ProductoY	Sacramento	Vizcarra, Federico T.
*	333445555	3	10.0	ProductoZ	Higueras	Nieto, Ramón K.
	333445555	3	10.0	ProductoZ	Higueras	Vizcarra, Federico T.
	333445555	10	10.0	Automatización	Santiago	Vizcarra, Federico T.
*	333445555	20	10.0	Reorganización	Higueras	Nieto, Ramón K.
	333445555	20	10.0	Reorganización	Higueras	Vizcarra, Federico T.

⋮

Figura 12.6 Resultado de aplicar la operación de REUNIÓN NATURAL a EMP_PROY1 y LUGARES_EMPS, con las tuplas espurias marcadas con *.

Es obvio que esta pauta informal deberá expresarse de manera más formal. En el capítulo 13 trataremos una condición formal, la llamada propiedad de reunión no aditiva (o sin pérdidas), con la que se garantizará que ciertas reuniones no producirán tuplas espurias.

12.1.5 *Análisis*

En las secciones 12.1.1 a 12.1.4 estudiamos informalmente situaciones que dan origen a esquemas de relación problemáticos y propusimos pautas informales para un buen diseño relacional. En lo que resta del capítulo expondremos conceptos y teorías formales con los que podremos definir con mayor precisión qué tan "bueno" o "malo" es un esquema de relación *individual*. Especificaremos varias formas normales para los esquemas de relación, y en el capítulo 13 presentaremos criterios adicionales para especificar que un *conjunto de esquemas de relación* constituye un buen esquema de base de datos relacional. También presentaremos algoritmos que aprovechan esta teoría para diseñar esquemas de bases de datos relacionales y definiremos formas normales adicionales que van más allá de la FNBC. Las formas normales que definiremos en este capítulo se basan en el concepto de dependencia funcional, que describiremos a continuación, en tanto que las formas normales que veremos en el capítulo 13 contemplan otros tipos de dependencias de los datos, que se denominan dependencias multivaluadas y dependencias de reunión.

12.2 Dependencias funcionales

El concepto individual más importante en el diseño de esquemas relacionales es el de dependencia funcional. En esta sección definiremos formalmente el concepto, y en la sección 12.3 veremos cómo da lugar a esquemas de relación en formas normales.

12.2.1 *Definición de dependencia funcional*

Una dependencia funcional es una restricción entre dos conjuntos de atributos de la base de datos. Suponga que nuestro esquema de base de datos relacional tiene n atributos $A_1, A_2, \ldots A_n$, y que toda la base de datos se describe con un solo esquema de relación **universal** $R = \{A_1, A_2, \ldots A_n\}$.[†] Con esto no implicamos que de hecho almacenaremos la base de datos como una sola tabla universal; únicamente vamos a usar este concepto para desarrollar la teoría formal de las dependencias de datos.[††]

Una **dependencia funcional**, denotada por $X \rightarrow Y$, entre dos conjuntos de atributos X y Y que son subconjuntos de R especifica una restricción sobre las posibles tuplas que podrían formar un ejemplar de relación r de R. La restricción dice que, para cualesquier dos tuplas t_1 y t_2 de r tales que $t_1[X] = t_2[X]$, debemos tener también $t_1[Y] = t_2[Y]$. Esto significa que los valores del componente Y de una tupla de r dependen de los valores del componente X, o están determinados por ellos; o bien, que los valores del componente X de una tupla **determinan** de manera única (o **funcionalmente**) los valores del componente Y. También decimos que hay una dependencia funcional de X a Y o que Y **depende funcionalmente** de X. Abreviaremos la dependencia funcional con DF. El conjunto de atributos X se denomina **miembro izquierdo** de la DF, y Y es el **miembro derecho**.

Así pues, X determina funcionalmente a Y en un esquema de relación R si y sólo si, siempre que dos tuplas $r(R)$ coincidan en su valor X, necesariamente deben coincidir en su valor Y. Observe que:

- Si una restricción de R dice que no puede haber más de una tupla con un valor X dado en cualquier ejemplar de relación $r(R)$ —esto es, X es una **clave candidata** de R— esto implica que $X \rightarrow Y$ para cualquier subconjunto de atributos Y de R.
- Si $X \rightarrow Y$ en R, esto no nos dice si $Y \rightarrow X$ en R o no.

Las dependencias funcionales son propiedades del significado o la **semántica** de los atributos. Aprovechamos nuestra comprensión de la semántica de los atributos de R —esto es, qué relación hay entre ellos— para especificar las dependencias funcionales que deben cumplirse en *todos* los estados de relación (extensiones) r de R. Siempre que la semántica de dos conjuntos de atributos de R indique que debe cumplirse una dependencia funcional, especificaremos esa dependencia como una restricción. Las extensiones de relación $r(R)$ que satisfagan las restricciones de dependencia funcional se denominarán **extensiones permitidas** (o **estados de relación permitidos**) de R, porque obedecen las restricciones de dependencia funcional. Por ende, la utilidad principal de las dependencias funcionales es describir mejor un esquema de relación R mediante la especificación de restricciones sobre sus atributos que deban cumplirse *siempre*.

Consideremos el esquema de relación EMP_PROY de la figura 12.3(b); a partir de la semántica de los atributos, sabemos que deben cumplirse las siguientes dependencias funcionales:

(a) NSS → NOMBREE
(b) NÚMEROP → {NOMBREPR, LUGARP}
(c) {NSS, NÚMEROP} → HORAS

Estas dependencias funcionales especifican que (a) el valor del número de seguro social de un empleado (NSS) determina de manera única el nombre de ese empleado (NOMBREE), (b) el valor del número de un proyecto (NÚMEROP) determina de manera única el nombre del proyecto (NOMBREPR) y su lugar (LUGARP), y (c) una combinación de valores de NSS y NÚMEROP determina de manera única el número de horas que el empleado trabaja en el proyecto cada semana (HORAS). Alternativamente, podemos decir que NOMBREE está determinado funcionalmente por (o depende funcionalmente de) NSS, o "dado un valor de NSS, conocemos el valor de NOMBREE", y así sucesivamente.

La figura 12.3 también presenta una notación diagramática para indicar las dependencias funcionales: cada DF se muestra como una línea horizontal. Los atributos del miembro izquierdo de la DF se conectan mediante líneas verticales a la línea horizontal que representa la DF. Los atributos del miembro derecho de la DF se conectan a la línea horizontal mediante flechas que apuntan a los atributos, como se aprecia en las figuras 12.3(a) y (b).

Una dependencia funcional es una *propiedad del esquema de relación* (intensión) R, y no de un ejemplar de relación permitido (extensión) r de R en particular. Por ello, una DF *no puede* inferirse automáticamente de una extensión de relación r dada, sino que debe definirla explícitamente alguien que conozca la semántica de los atributos de R. Por ejemplo,

[†]Este concepto de relación universal adquirirá importancia cuando analicemos los algoritmos para diseñar bases de datos en el capítulo 13.

[††]Esta suposición significa que todos los atributos de la base de datos deben tener un *nombre distinto*. En el capítulo 6 pusimos como prefijos los nombres de relación a los nombres de atributos para lograr la unicidad cada vez que los atributos de diferentes relaciones tenían el mismo nombre.

IMPARTIR

PROFESOR	CURSO	TEXTO
Silva	Estructuras de datos	Bartram
Silva	Gestión de datos	Al-Nour
Heras	Compiladores	Hoffman
Bravo	Estructuras de datos	Augenthaler

Figura 12.7 La relación IMPARTIR.

la figura 12.7 ilustra un ejemplar de relación específico del esquema de relación IMPARTIR. Aunque a primera vista podríamos sentirnos tentados a decir que TEXTO → CURSO, no podremos confirmar esto a menos que sepamos que se cumple para todos los posibles estados de relación de IMPARTE. Por otro lado, basta con presentar un solo contraejemplo para demostrar que no existe una dependencia funcional. Por ejemplo, del hecho de que 'Silva' imparte tanto 'Estructuras de datos' como 'Gestión de datos' podemos concluir que PROFESOR no determina funcionalmente CURSO. Denotamos esto con PROFESOR $\overset{X}{\rightarrow}$ CURSO. A partir de la figura 12.7 podemos decir también que CURSO $\overset{X}{\rightarrow}$ TEXTO.

12.2.2 *Reglas de inferencia para las dependencias funcionales*

Denotamos con F el conjunto de dependencias funcionales que se especifican sobre el esquema de relación R. Por lo regular, el diseñador del esquema especifica las dependencias funcionales que son *semánticamente obvias*, pero es común que muchas otras dependencias funcionales se cumplan en *todos* los ejemplares de relación permitidos que satisfagan las dependencias de F. El conjunto de todas estas dependencias funcionales se denomina **cerradura** de F y se denota con F^+. Por ejemplo, suponga que especificamos el siguiente conjunto F de dependencias funcionales obvias sobre el esquema de relación de la figura 12.3(a):

F = { NSS → {NOMBREE, FECHAN, DIRECCIÓN, NÚMEROD},
NÚMEROD → {NOMBRED, NSSGTED}}

Podemos **inferir** las siguientes dependencias funcionales adicionales a partir de F:

NSS → {NOMBRED, NSSGTED},
NSS → NSS,
NÚMEROD → NOMBRED

Una DF $X \rightarrow Y$ se **infiere de** un conjunto de dependencias F especificado sobre R si se cumple $X \rightarrow Y$ en *todo* estado de relación r que sea una extensión permitida de R; es decir, siempre que r satisfaga todas las dependencias de F, $X \rightarrow Y$ también se cumpla en r. La cerradura F^+ de F es el conjunto de todas las dependencias funcionales que pueden inferirse de F. Para determinar una forma sistemática de inferir dependencias, tendremos que descubrir un conjunto de **reglas de inferencia** que pueda servir para deducir nuevas dependencias a partir de un conjunto dado de dependencias. A continuación veremos algunas de estas reglas de inferencia. Con la notación $F \not\models X \rightarrow Y$ denotaremos que la dependencia funcional $X \rightarrow Y$ se infiere del conjunto de dependencias funcionales F.

En el análisis que sigue, usaremos una notación abreviada para referirnos a las dependencias funcionales. Por comodidad, concatenaremos las variables de atributos y omitiremos las comas. Así pues, la DF $\{X,Y\} \rightarrow Z$ se abreviará como $XY \rightarrow Z$, y la DF $\{X,Y,Z\} \rightarrow \{U,V\}$ se abreviará como $XYZ \rightarrow UV$. Las seis reglas que siguen (RI1 a RI6) son reglas de inferencia bien conocidas para las dependencias funcionales:

(RI1) (Regla reflexiva) Si $X \supseteq Y$, entonces $X \rightarrow Y$.
(RI2) (Regla de aumento†) $\{X \rightarrow Y\} \models XZ \rightarrow YZ$.
(RI3) (Regla transitiva) $\{X \rightarrow Y, Y \rightarrow Z\} \models X \rightarrow Z$.
(RI4) (Regla de descomposición (o proyectiva)) $\{X \rightarrow YZ\} \models X \rightarrow Y$.
(RI5) (Regla de unión (o aditiva)) $\{X \rightarrow Y, X \rightarrow Z\} \models X \rightarrow YZ$.
(RI6) (Regla pseudotransitiva) $\{X \rightarrow Y, WY \rightarrow Z\} \models WX \rightarrow Z$.

La regla reflexiva (RI1) dice que un conjunto de atributos siempre se determina a sí mismo, lo cual es obvio. La regla de aumento (RI2) dice que añadir el mismo conjunto de atributos a los dos miembros, izquierdo y derecho, de una dependencia produce otra dependencia válida. Según RI3, las dependencias funcionales son transitivas. La regla de descomposición (RI4) dice que podemos quitar atributos del miembro derecho de una dependencia; la aplicación repetida de esta regla puede descomponer la DF $X \rightarrow \{A_1, A_2, ..., A_n\}$ en el conjunto de dependencias $\{X \rightarrow A_1, X \rightarrow A_2, ..., X \rightarrow A_n\}$. La regla de unión (RI5) nos permite hacer lo opuesto: combinar un conjunto de dependencias $\{X \rightarrow A_1, X \rightarrow A_2, ..., X \rightarrow A_n\}$ en una sola DF $X \rightarrow \{A_1, A_2, ..., A_n\}$.

Todas las reglas de inferencia anteriores pueden demostrarse a partir de la definición de dependencia funcional, sea por demostración directa o **por contradicción**. Una demostración por contradicción (o reducción al absurdo) supone que la regla no se cumple y muestra que esto no es posible. A continuación demostraremos que las tres primeras reglas (RI1 a RI3) son válidas. La segunda demostración es por contradicción.

DEMOSTRACIÓN DE R11
Suponga que $X \supseteq Y$ y que existen dos tuplas t_1 y t_2 en algún ejemplar de relación r de R tales que $t_1[X] = t_2[X]$. Entonces, $t_1[Y] = t_2[Y]$ porque $X \supseteq Y$; por tanto, debe cumplirse $X \rightarrow Y$ en r.

DEMOSTRACIÓN DE R12 (POR CONTRADICCIÓN)
Suponga que $X \rightarrow Y$ se cumple en un ejemplar de relación r de R, pero que $XZ \rightarrow YZ$ no se cumple. En tal caso, deben existir dos tuplas t_1 y t_2 en r tales que (1) $t_1[X] = t_2[X]$, (2) $t_1[Y] = t_2[Y]$, (3) $t_1[XZ] = t_2[XZ]$ y (4) $t_1[YZ] \neq t_2[YZ]$. Esto no es posible porque a partir de (1) y (3) se deduce (5) $t_1[Z] = t_2[Z]$, y a partir de (2) y (5) se deduce (6) $t_1[YZ] = t_2[YZ]$, lo que contradice (4).

DEMOSTRACIÓN DE R13
Suponga que se cumplen (1) $X \rightarrow Y$ y (2) $Y \rightarrow Z$ en una relación r. Entonces, para cualesquier dos tuplas t_1 y t_2 en r tales que $t_1[X] = t_2[X]$, debemos tener (3) $t_1[Y] = t_2[Y]$ (por la suposición (1)), y por tanto también debemos tener (4) $t_1[Z] = t_2[Z]$, (por (3) y a partir de la suposición (2)); por tanto, se debe cumplir $X \rightarrow Z$ en r.

†La regla de aumento también puede expresarse como $\{X \rightarrow Y\} \models XZ \rightarrow Y$; es decir, aumentar los atributos del miembro izquierdo de una DF producirá otra DF válida.

Con argumentos similares podemos demostrar las reglas de inferencia RI4 a RI6 y cualesquier otras reglas de inferencia válidas. Sin embargo, una forma más simple de demostrar que una regla de inferencia para dependencias funcionales es válida consiste en demostrarla mediante reglas de inferencia cuya validez ya se ha demostrado. Por ejemplo, podemos demostrar RI4 a RI6 a partir de RI1 a RI3, como sigue:

DEMOSTRACIÓN DE R14

1. $X \rightarrow YZ$ (dado).
2. $YZ \rightarrow Y$ (por RI1 y sabiendo que $YZ \supseteq Y$).
3. $X \rightarrow Y$ (por RI3 en 1 y 2).

DEMOSTRACIÓN DE R15

1. $X \rightarrow Y$ (dado).
2. $X \rightarrow Z$ (dado).
3. $X \rightarrow XY$ (por RI2 en 1 aumentándolo con X; observe que $XX = X$).
4. $XY \rightarrow YZ$ (por RI2 en 2 aumentándolo con Y).
5. $X \rightarrow YZ$ (por RI3 en 3 y 4).

DEMOSTRACIÓN DE R16

1. $X \rightarrow Y$ (dado).
2. $WY \rightarrow Z$ (dado).
3. $WX \rightarrow WY$ (con RI2 en 1 aumentándolo con W).
4. $WX \rightarrow Z$ (con RI3 en 3 y 2).

Armstrong (1974) demostró que las reglas de inferencia RI1 a RI3 son **correctas** y **completas**. Con correctas queremos decir que, dado un conjunto de dependencias funcionales F especificado sobre un esquema de relación R, cualquier dependencia que podamos inferir de F con RI1, RI2 y RI3 se cumplirá en todos los estados de relación r de R que satisfagan las *dependencias* de F. Con completas queremos decir que el empleo repetido de RI1, RI2 y RI3 para inferir dependencias hasta que no sea posible inferir más dependencias producirá el conjunto completo de *todas las dependencias posibles* que se pueden inferir de F. En otras palabras, el conjunto de dependencias F^+, al que llamamos cerradura de F, se puede determinar a partir de F utilizando sólo las reglas de inferencia RI1, RI2 y RI3. Las reglas de inferencia RI1 a RI3 se conocen como **reglas de inferencia de Armstrong.**[†]

Por lo regular, los diseñadores de bases de datos especifican primero el conjunto de dependencias funcionales F que se pueden determinar sin dificultad a partir de la semántica de los atributos de R. Luego, pueden usar las reglas de inferencia de Armstrong para inferir dependencias funcionales adicionales que también se cumplirán en R. Una forma sistemática de determinar estas dependencias funcionales adicionales consiste en determinar primero todos y cada uno de los conjuntos de atributos X que aparezcan como miembro izquierdo de alguna dependencia funcional en F y después usar las reglas de inferencia

[†]En realidad se conocen como **axiomas de Armstrong**, aunque no son axiomas en el sentido matemático. En términos estrictos, los axiomas son las dependencias funcionales de F, pues suponemos que son correctas, y RI1, RI2 y RI3 son las reglas de inferencia para deducir nuevas dependencias funcionales.

de Armstrong para determinar el conjunto de todos los atributos que dependan de X. Así, para cada conjunto de atributos X, determinamos el conjunto X^+ de atributos determinados funcionalmente por X; X^+ se denomina **cerradura de X bajo F.** Podemos usar el algoritmo 12.1 para calcular X^+.

ALGORITMO 12.1 Determinar X^+, la cerradura de X bajo F.

```
X+ := X;
repetir
        viejoX+ := X+;
        para cada dependencia funcional Y→ Z en F hacer
        si Y⊆ X+ entonces X+ := X+ ∪ Z;
hasta que (viejoX+ = X+);
```

El algoritmo 12.1 comienza por igualar X^+ a todos los atributos de X. Por RI1, sabemos que todos estos atributos dependen funcionalmente de X. En virtud de las reglas de inferencia RI3 y RI4, añadimos atributos a X^+, empleando cada una de las dependencias funcionales en F. Seguimos pasando por todas las dependencias en F (el ciclo repetir) hasta que no se añadan más atributos a X^+ durante un recorrido completo de las dependencias en F. Por ejemplo, consideremos el esquema de relación EMP_PROY de la figura 12.3(b); por la semántica de los atributos, especificamos el siguiente conjunto F de dependencias funcionales que deben cumplirse en EMP_PROY:

$$F = \{ \text{NSS} \rightarrow \text{NOMBREE},\ \text{NÚMEROP} \rightarrow \{\text{NOMBREPR, LUGARP}\},\ \{\text{NSS, NÚMEROP}\} \rightarrow \text{HORAS} \}$$

Con el algoritmo 12.1 podemos calcular los siguientes conjuntos de cerradura respecto a F:

$$\{ \text{NSS} \}^+ = \{ \text{NSS, NOMBREE} \}$$
$$\{ \text{NÚMEROP} \}^+ = \{ \text{NÚMEROP, NOMBREPR, LUGARP} \}$$
$$\{ \text{NSS, NÚMEROP} \}^+ = \{ \text{NSS, NÚMEROP, NOMBREE, NOMBREPR, LUGARP, HORAS} \}$$

12.2.3 *Equivalencia de conjuntos de dependencias funcionales*★

En esta sección analizaremos la equivalenca de dos conjuntos de dependencias funcionales. Primero, daremos algunas definiciones preliminares. Un conjunto de dependencias funcionales E está **cubierto por** un conjunto de dependencias funcionales F —o bien, se dice que F **cubre** a E— si toda DF en E también está en F^+; es decir, E está cubierto si toda dependencia en E puede inferirse a partir de F. Dos conjuntos de dependencias funcionales E y F son **equivalentes** si $E^+ = F^+$. Así pues, la equivalencia significa que todas las DF en E se pueden inferir de F y que todas las DF en F se pueden inferir de E; esto es, E es equivalente a F si se cumple que E cubre a F y que F cubre a E.

Podemos determinar si F cubre a E calculando X^+ respecto a F para cada DF $X \rightarrow Y$ en E, y comprobando después que este X^+ incluya los atributos en Y. Si esto se cumple para *todas* las DF en E, entonces F cubre a E. Determinamos si E y F son equivalentes verificando que E cubra a F y que F cubra a E.

12.2.4 *Conjuntos mínimos de dependencias funcionales*★

Un conjunto de dependencias funcionales F es **mínimo** si satisface las siguientes condiciones:

1. Toda dependencia en F tiene un solo atributo en su miembro derecho.
2. No podemos quitar ninguna dependencia de F y seguir teniendo un conjunto de dependencias equivalente a F.
3. No podemos reemplazar ninguna dependencia $X \rightarrow A$ en F por una dependencia $Y \rightarrow A$, donde Y es un subconjunto propio de X, y seguir teniendo un conjunto de dependencias equivalente a F.

Podemos concebir un conjunto mínimo de dependencias como un conjunto de dependencias que está en una forma estándar o canónica sin redundancias. Las condiciones 2 y 3 garantizan que no habrá redundancias en las dependencias, y la condición 1 asegura que toda dependencia está en una forma canónica con un solo atributo en el miembro derecho. Una **cobertura mínima** de un conjunto de dependencias funcionales F es un conjunto mínimo de dependencias $F_{mín}$ que es equivalente a F. Desafortunadamente, un conjunto de dependencias funcionales puede tener varias coberturas mínimas. Siempre podemos hallar *por lo menos una* cobertura mínima $F_{mín}$ para cualquier conjunto de dependencias F (véase la Sec. 13.1.2).

12.3 Formas normales basadas en claves primarias

En esta sección estudiaremos el proceso de normalización y definiremos las tres primeras formas normales para los esquemas de relación. Las definiciones de segunda y tercera formas normales que aquí presentamos se basan en las dependencias funcionales y claves primarias de un esquema de relación. Veremos cómo se desarrollaron históricamente estas formas normales y la idea intuitiva en que se apoyaron. En la sección 12.4 presentaremos definiciones más generales de estas formas normales, que tienen en cuenta *todas las claves candidatas* de una relación y *no sólo la clave primaria*. En la sección 12.5 definiremos la forma normal de Boyce-Codd (FNBC), y en el capítulo 13 definiremos otras formas normales que se basan en otros tipos de dependencias con respecto a los datos.

Primero, en la sección 12.3.1, analizaremos informalmente qué son las formas normales y qué motivos hubo para su creación; además, recordaremos algunas de las definiciones del capítulo 6 que vamos a necesitar aquí. En seguida, explicaremos la primera forma normal (1FN) en la sección 12.3.2. En las secciones 12.3.3 y 12.3.4 presentaremos definiciones de la segunda y tercera formas normales (2FN y 3FN), respectivamente, las cuales se basan en las claves primarias.

12.3.1 *Introducción a la normalización*

En el proceso de normalización, según la propuesta original de Codd (1972*a*), se somete un esquema de relación a una serie de pruebas para "certificar" si pertenece o no a una cierta **forma normal**. En un principio, Codd propuso tres formas normales, a las cuales llamó primera, segunda y tercera formas normales. Posteriormente, Boyce y Codd propusieron una

definición más estricta de 3FN, a la que se conoce como forma normal de Boyce-Codd. Todas estas formas normales se basan en las dependencias funcionales entre los atributos de una relación. Más adelante se propusieron una cuarta forma normal (4FN) y una quinta (5FN), con fundamento en los conceptos de dependencias multivaluadas y dependencias de reunión, respectivamente; éstas se estudiarán en el capítulo 13.

La **normalización de los datos** puede considerarse como un proceso durante el cual los esquemas de relación insatisfactorios se descomponen repartiendo sus atributos entre esquemas de relación más pequeños que poseen propiedades deseables. Un objetivo del proceso de normalización original es garantizar que no ocurran las anomalías de actualización que vimos en la sección 12.1.2. Las formas normales proveen a los diseñadores de bases de datos lo siguiente:

- Un marco formal para analizar los esquemas de relación con base en sus claves y en las dependencias funcionales entre sus atributos.
- Una serie de pruebas que pueden efectuarse sobre esquemas de relación individuales de modo que la base de datos relacional pueda **normalizarse** hasta el grado deseado. Cuando una prueba falla, la relación que provoca el fallo debe descomponerse en relaciones que individualmente satisfagan las pruebas de normalización.

Las formas normales, consideradas *aparte* de otros factores, no garantizan un buen diseño de base de datos. En general, no basta con comprobar por separado que cada esquema de relación de la base de datos esté en, digamos, FNBC o 3FN. Más bien, el proceso de normalización por descomposición debe confirmar también la existencia de propiedades adicionales que los esquemas relacionales, en conjunto, deben poseer. Dos de estas propiedades son:

- La propiedad de reunión sin pérdida o reunión no aditiva, que garantiza que no se presentará el problema de las tuplas espurias (véase la Sec. 12.1.4).
- La propiedad de conservación de las dependencias, que asegura que todas las dependencias funcionales estén representadas en algunas de las relaciones individuales resultantes.

Dejaremos para el capítulo 13 la exposición de los conceptos formales y de las técnicas que garantizan las dos propiedades mencionadas. En esta sección nos concentraremos en un *análisis intuitivo* del proceso de normalización. Cabe señalar que las formas normales mencionadas en esta sección no son las únicas posibles. Es posible definir formas normales adicionales para satisfacer otros criterios deseables, basados en tipos de restricciones adicionales. Las formas normales hasta FNBC se definen considerando sólo las restricciones de dependencia funcional y de clave, en tanto que la 4FN considera una restricción adicional denominada dependencia multivaluada, y la 5FN considera una restricción más llamada dependencia de reunión. La utilidad práctica de las formas normales queda en entredicho cuando las restricciones en las que se basan son difíciles de entender o de detectar por parte de los diseñadores de bases de datos y usuarios que deben descubrir estas restricciones.

Otro punto que merece la pena destacar es que los diseñadores de bases de datos *no tienen que* normalizar hasta la forma normal más alta posible. Las relaciones pueden dejarse en formas normales inferiores por razones de rendimiento, como las que examinamos al final de la sección 12.1.2.

Antes de continuar, recordemos las definiciones de claves de un esquema de relación que dimos en el capítulo 6. Una **superclave** de un esquema de relación $R = \{A_1, A_2, ..., A_n\}$ es un conjunto de atributos $S \subseteq R$ con la propiedad de que no habrá un par de tuplas t_1 y t_2 en ningún estado de relación permitido r de R tal que $t_1[S] = t_2[S]$. Una **clave** K es una superclave con la propiedad adicional de que la eliminación de cualquier atributo de K hará que K deje de ser una superclave. La diferencia entre una clave y una superclave es que la primera tiene que ser "mínima"; esto es, si tenemos una clave $K = \{A_1, A_2, ..., A_k\}$, entonces $K - A_i$ no es una clave para $1 \leq i \leq k$. En la figura 12.1 {NSS} es una clave de EMPLEADO, y {NSS}, {NSS, NOMBREE}, {NSS, NOMBREE, FECHAN}, etc., son todas superclaves.

Si un esquema de relación tiene más de una clave "mínima", todas son **claves candidatas**. Una de las claves candidatas se designa *arbitrariamente* como **clave primaria**, y las demás se denominan claves secundarias. Todo esquema de relación debe tener una clave primaria. En la figura 12.1 {NSS} es la única clave candidata de EMPLEADO, así que también es la clave primaria.

Un atributo del esquema de relación R se denomina **atributo primo** de R si es miembro de *cualquier clave* de R. Un atributo es **no primo** si no es un atributo primo; es decir, si no es miembro de ninguna clave candidata.. En la figura 12.1 tanto NSS como NÚMEROP son atributos primos de TRABAJA_EN, y los demás atributos de TRABAJA_EN son no primos.

Ahora presentaremos las tres primeras formas normales: 1FN, 2FN y 3FN. Codd (1972*a*) las propuso como una secuencia para alcanzar el estado deseable de relaciones 3FN avanzando por los estados intermedios de 1FN y 2FN, si es necesario.

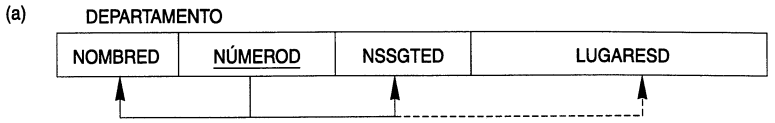
12.3.2 *Primera forma normal (1FN)*

La **primera forma normal** se considera ahora parte de la definición formal de relación; históricamente, se definió para prohibir los atributos multivaluados, los atributos compuestos y sus combinaciones. Establece que los dominios de los atributos deben incluir sólo *valores atómicos* (simples, indivisibles) y que el valor de cualquier atributo en una tupla debe ser un *valor individual* proveniente del dominio de ese atributo. Así pues, 1FN prohíbe tener un conjunto de valores, una tupla de valores o una combinación de ambos como valor de un atributo para una *tupla individual*. En otras palabras, 1FN prohíbe las "relaciones dentro de relaciones" o las "relaciones como atributos de tuplas". Los únicos valores de atributos que permite 1FN son **valores atómicos** (o **indivisibles**).

Consideremos el esquema de relación DEPARTAMENTO de la figura 12.1, cuya clave primaria es NÚMEROD, y supongamos que la extendemos al incluir el atributo LUGARESD que aparece en líneas punteadas. Suponemos que cada departamento puede tener *varios* lugares. En la figura 12.8 se muestra el esquema DEPARTAMENTO y un ejemplo de extensión. Es evidente que no está en 1FN porque LUGARESD no es un atributo atómico, como puede verse en la primera tupla de la figura 12.8(b). Hay dos formas de considerar el atributo LUGARESD:

- El dominio de LUGARESD contiene valores atómicos, pero algunas tuplas pueden tener un conjunto de esos valores. En este caso, NSS $\twoheadrightarrow$ LUGARESD.
- El dominio de LUGARESD contiene conjuntos de valores y por tanto no es atómico. En este caso, NSS $\rightarrow$ LUGARESD, porque cada conjunto se considera un miembro individual del dominio del atributo.†

†En este caso podemos considerar que el dominio de LUGARESD es el **conjunto potencia** del conjunto de lugares individuales; esto es, el dominio consiste en *todos los posibles subconjuntos* del conjunto de lugares individuales.



(b) DEPARTAMENTO

NOMBRED	NÚMEROD	NSSGTED	LUGARESD
Investigación	5	333445555	(Belén, Sacramento, Higueras)
Administración	4	987654321	(Santiago)
Dirección	1	888665555	(Higueras)

(c) DEPARTAMENTO

NOMBRED	NÚMEROD	NSSGTED	LUGARESD
Investigación	5	333445555	Belén
Investigación	5	333445555	Sacramento
Investigación	5	333445555	Higueras
Administración	4	987654321	Santiago
Dirección	1	888665555	Higueras

Figura 12.8 Normalización a 1FN. (a) Esquema de relación que no está en 1FN. (b) Ejemplo de ejemplar de relación. (c) Relación 1FN con redundancia.

En cualquier caso, la relación DEPARTAMENTO de la figura 12.8 no está en 1FN; de hecho, ni siquiera califica como una relación, según nuestra definición de la sección 6.1. Para normalizarla a relaciones 1FN, dividimos sus atributos entre las dos relaciones DEPARTAMENTO y LUGARES_DEPTOS de la figura 12.2. La idea es eliminar el atributo LUGARESD que viola 1FN y colocarlo en una relación aparte LUGARES_DEPTOS junto con la clave primaria NÚMEROD de DEPARTAMENTO. La clave primaria de esta relación es la combinación {NÚMEROD, LUGARD}, como se aprecia en la figura 12.2. Hay una tupla distinta en LUGARES_DEPTOS por *cada ubicación* de un departamento. El atributo LUGARESD se quita de la relación DEPARTAMENTO de la figura 12.8, descomponiendo la relación que no es 1FN en las dos relaciones 1FN DEPARTAMENTO y LUGARES_DEPTOS de la figura 12.2.

Observe que una segunda manera de normalizar a 1FN es tener una tupla en la relación DEPARTAMENTO original por cada ubicación de un DEPARTAMENTO, como se muestra en la figura 12.8(c). En este caso, la clave primaria se convierte en la combinación {NÚMEROD, LUGARD} y hay redundancia en las tuplas. La primera solución es mejor porque no padece este problema de redundancia. De hecho, si elegimos la segunda solución, experimentará más descomposiciones durante los pasos de normalización subsecuentes hasta dar la primera solución.

La primera forma normal también prohíbe los atributos compuestos que por sí mismos son multivaluados. Estos se denominan **relaciones anidadas** porque cada tupla puede tener una relación *dentro de sí*. La figura 12.9 muestra cómo podría representarse una relación EMP_PROY si se permitiera la anidación. Cada tupla representa una entidad empleado, y una relación PROYS(NÚMEROP, HORAS) *dentro de cada tupla* representa los proyectos del

empleado y las horas por semana que trabaja en cada proyecto. El esquema de la relación EMP_PROY podría representarse así:

EMP_PROY(NSS, NOMBREE, {PROYS(NÚMEROP, HORAS)})

Las llaves de conjunto {} identifican el atributo PROYS como multivaluado, y listamos los atributos componentes que forman PROYS entre paréntesis. Resulta interesante que investigaciones recientes sobre el modelo relacional intenten ahora que las relaciones anidadas puedan ser válidas, y formalizarlas, lo que por principio no era posible con la 1FN (véase la Sec. 21.6.2).

Adviértase que NSS es la clave primaria de la relación EMP_PROY en las figuras 12.9(a) y (b), mientras que NÚMEROP es la clave primaria **parcial** de cada relación anidada; esto es, dentro de cada tupla, la relación anidada debe tener valores únicos de NÚMEROP. Para normalizar esta relación a 1FN, pasamos los atributos de la relación anidada a una nueva relación y **propagamos la clave primaria** en ella; la clave primaria de la nueva relación combinará la clave parcial con la clave primaria de la relación original. La descomposición y la propagación de la clave primaria producirán los esquemas que ilustramos en la figura 12.9(c).

Este procedimiento puede aplicarse recursivamente a una relación con anidación de múltiples niveles para **desanidar** la relación y producir un conjunto de relaciones 1FN. Como veremos en el capítulo 13, si restringimos las relaciones a 1FN daremos pie a los problemas asociados a las dependencias multivaluadas y 4FN.

12.3.3 *Segunda forma normal (2FN)*

La segunda forma normal se basa en el concepto de dependencia funcional total. Una dependencia funcional $X \rightarrow Y$ es una **dependencia funcional total** si la eliminación de cualquier atributo A de X hace que la dependencia deje de ser válida; es decir, para cualquier atributo $A \in X$, $(X - \{A\}) \not\rightarrow Y$. Una dependencia funcional $X \rightarrow Y$ es una **dependencia parcial** si es posible eliminar un atributo $A \in X$ de X y la dependencia sigue siendo válida; es decir, para algún $A \in X$, $(X - \{A\}) \rightarrow Y$. En la figura 12.3(b), {NSS, NÚMEROP} → HORAS es una dependencia total (no se cumplen ni NSS → HORAS ni NÚMEROP → HORAS). No obstante, la dependencia {NSS, NÚMEROP} → NOMBREE es parcial porque se cumple que NSS → NOMBREE.

Un esquema de relación *R* está en **2FN** si todo atributo no primo *A* en *R depende funcionalmente de manera total* de la clave primaria de *R*. La relación EMP_PROY de la figura 12.3(b) está en 1FN pero no en 2FN. El atributo no primo NOMBREE viola 2FN debido a df2, y lo mismo sucede con los atributos no primos NOMBREPR y LUGARP debido a df3. Las dependencias funcionales df2 y df3 hacen que NOMBREE, NOMBREPR y LUGARP dependan parcialmente de la clave primaria {NSS, NÚMEROP} de EMP_PROY, violándose así 2FN.

Si un esquema de relación no está en 2FN, se le puede normalizar a varias relaciones 2FN en las que los atributos no primos estén asociados sólo a la parte de la clave primaria de la que dependen funcionalmente de manera total. Así, las dependencias funcionales df1, df2 y df3 de la figura 12.3(b) originan la descomposición de EMP_PROY en los tres esquemas de relación EP1, EP2 y EP3 que ilustra la figura 12.10(a), cada uno de los cuales está en 2FN. Es evidente que las relaciones EP1, EP2 y EP3 carecen de las anomalías de actualización de las que adolece EMP_PROY en la figura 12.3(b).

(a) **EMP_PROY**

NSS	NOMBREE	PROYS	
		NÚMEROP	HORAS

(b) **EMP_PROY**

NSS	NOMBREE	NÚMEROP	HORAS
123456789	Silva, José B.	1	32.5
		2	7.5
666884444	Nieto, Ramón K.	3	40.0
453453453	Esparza, Josefa A.	1	20.0
		2	20.0
333445555	Vizcarra, Federico T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zapata, Alicia J.	30	30.0
		10	10.0
987987987	Jabbar, Amhed V.	10	35.0
		30	5.0
987654321	Valdés Jazmín S.	30	20.0
		20	15.0
888665555	Botello, Jaime E.	20	nulo

(c) **EMP_PROY**

NSS	NOMBREE

EMP_PROY

NSS	NÚMEROP	HORAS

Figura 12.9 Normalización de relaciones anidadas a 1FN. (a) Esquema de la relación EMP_PROY con una "relación anidada" PROYS dentro de EMP_PROY. (b) Ejemplo de extensión de la relación EMP_PROY con relaciones anidadas dentro de cada tupla. (c) Descomposición de EMP_PROY a relaciones 1FN por migración de la clave primaria.

12.3.4 *Tercera forma normal (3FN)*

La tercera forma normal se basa en el concepto de dependencia transitiva. Una dependencia funcional $X \rightarrow Y$ en un esquema de relación *R* es una **dependencia transitiva** si existe un conjunto de atributos Z que *no sea un subconjunto* de cualquier clave† de *R*, y se cumplen

†Ésta es la definición general de dependencia transitiva. Como sólo nos ocupamos de las claves primarias en esta sección, permitimos dependencias transitivas en las que X es la clave primaria, pero Z puede ser (un subconjunto de) una clave candidata.

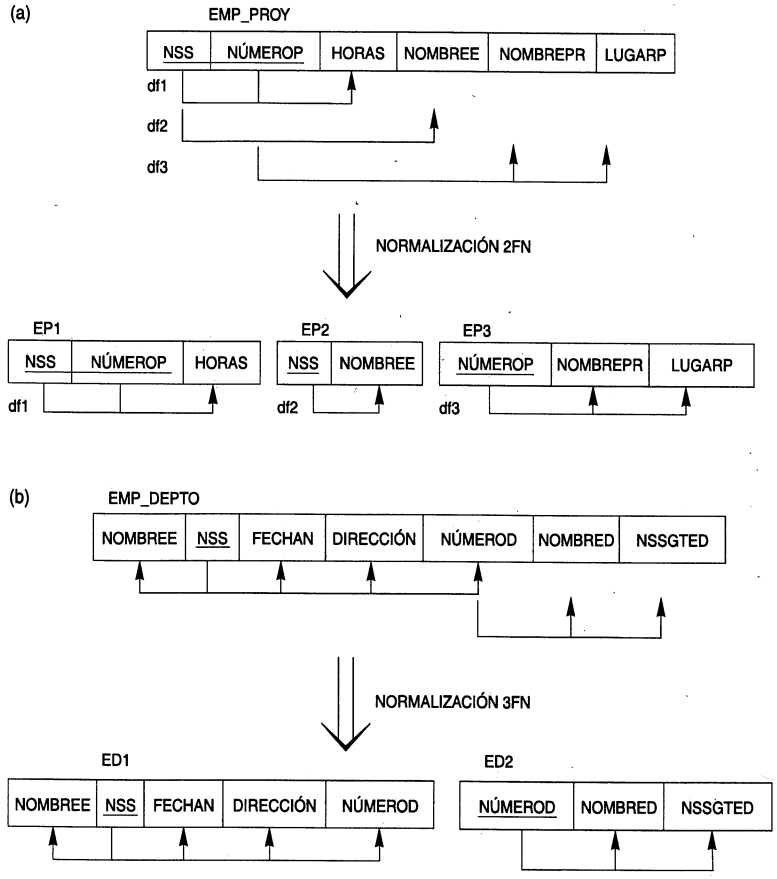


Figura 12.10 El proceso de normalización. (a) Normalización de EMP_PROY a relaciones 2FN. (b) Normalización de EMP_DEPTO a relaciones 3FN.

tanto $X \rightarrow Z$ como $Z \rightarrow Y$. La dependencia NSS → NSSGTED es transitiva a través de NÚMEROD de EMP_DEPTO en la figura 12.3(a), porque se cumplen las dos dependencias NSS → NÚMEROD y NÚMEROD → NSSGTED y NÚMEROD no es un subconjunto de la clave de EMP_DEPTO. Intuitivamente, podemos ver que en EMP-DEPTO no es deseable la dependencia de NSSGTED con respecto a NÚMEROD porque NÚMEROD no es una clave de EMP_DEPTO.

De acuerdo con la definición original de Codd, un esquema de relación R está en 3FN si está en 2FN y ningún atributo no primo de R depende transitivamente de la clave primaria. El esquema de relación EMP_DEPTO de la figura 12.3(a) está en 2FN, pues no existen dependencias parciales de una clave. Sin embargo, no está en 3FN debido a que NSSGTED (y también NOMBRED) dependen transitivamente de NSS a través de NÚMEROD. Podemos normalizar EMP_DEPTO descomponiéndolo en los dos esquemas de relación 3FN ED1 y ED2 que aparecen en la figura 12.10(b). Intuitivamente, vemos que ED1 y ED2 representan hechos independientes acerca

de las entidades empleados y departamentos. Una operación de REUNIÓN NATURAL aplicada a ED1 y ED2 recuperará la relación original EMP_DEPTO sin generar tuplas espurias.

12.4 Definiciones generales de la segunda y tercera formas normales

En general, queremos diseñar nuestros esquemas de relaciones de modo que no tengan dependencias parciales ni transitivas, ya que estos tipos de dependencias provocan las anomalías de actualización que vimos en la sección 12.1.2. Según las definiciones de segunda y tercera forma normal expuestas en la sección 12.3, los pasos para la normalización a relaciones 3FN prohíben las dependencias parciales y transitivas sobre la *clave primaria*. Estas definiciones, empero, no tienen en cuenta otras claves candidatas de una relación, si existen.

En esta sección presentaremos las definiciones más generales de 2FN y 3FN que tienen en cuenta *todas* las claves candidatas de una relación. Cabe señalar que esto no afecta la definición de 1FN, ya que es independiente de las claves y de las dependencias funcionales. Usaremos las *definiciones generales* de los atributos primos, de las dependencias funcionales parciales y totales y de las dependencias transitivas que explicamos antes y que consideran todas las claves candidatas de una relación.

12.4.1 *Definición general de segunda forma normal (2FN)*

Un esquema de relación R está en **segunda forma normal** (2FN) si ningún atributo no primo A de R depende parcialmente de *cualquier clave* de R.[†] Consideremos el esquema de relación LOTES que aparece en la figura 12.11(a) y que describe terrenos a la venta en diversos municipios de un estado. Supongamos que hay dos claves candidatas: ID_PROPIEDAD y {NOMBRE_MUNIC, NÚM_LOTE}; es decir, los NÚM_LOTE son únicos sólo dentro de cada municipio, pero los identificadores de propiedad (ID_PROPIEDAD) son únicos para todo el estado, sin importar el municipio.

Con base en las dos claves candidatas ID_PROPIEDAD y {NOMBRE_MUNIC, NÚM_LOTE}, sabemos que las dependencias funcionales df1 y df2 de la figura 12.11(a) sí se cumplen. Escogemos ID_PROPIEDAD como clave primaria, y por ello está subrayada en la figura 12.11(a). Supongamos que también se cumplen las siguientes dos dependencias funcionales en LOTES:

df3: NOMBRE_MUNIC→ TASA_FISCAL
df4: ÁREA → PRECIO

En español, la dependencia df3 dice que la tasa fiscal es fija para un municipio dado (no varía de un lote a otro dentro del mismo municipio), y df4 indica que el precio de un lote lo determina su área sin importar en qué municipio se encuentre (supongamos que éste es el precio del lote para fines fiscales).

El esquema de relación LOTES viola la definición general de 2FN porque TASA_FISCAL depende parcialmente de la clave candidata {NOMBRE_MUNIC, NÚM_LOTE} debido a df3. Para normalizar LOTES a 2FN, la descomponemos en las dos relaciones LOTES1 y LOTES2 que se muestran en la figura 12.11(b). Construimos LOTES1 eliminando de LOTES el atributo

[†]Esta definición puede expresarse como sigue: un esquema de relación R está en 2NF si todo atributo no primo A de R depende funcionalmente de manera total de *toda clave* de R.

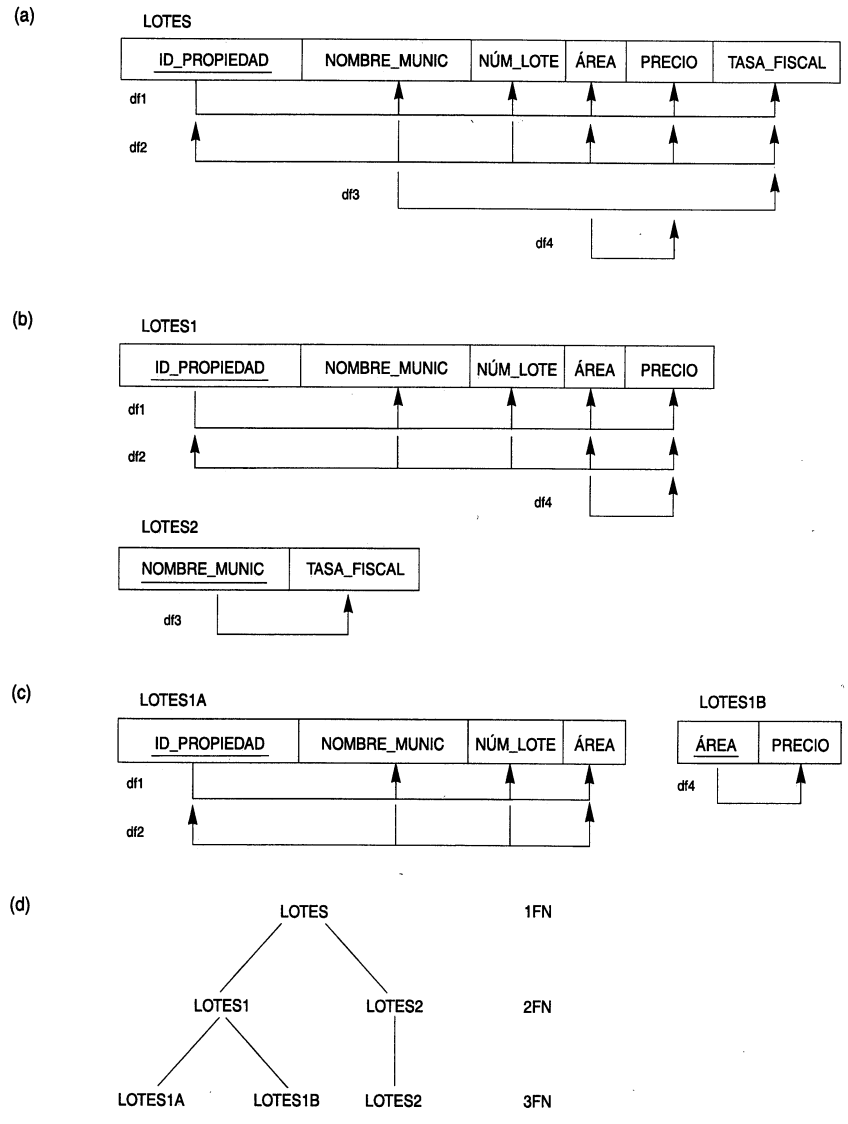


Figura 12.11 Normalización a 2FN y 3FN. (a) El esquema de relación LOTES y sus dependencias funcionales df1 a df4. (b) Descomposición de LOTES en las relaciones 2FN LOTES1 y LOTES2. (c) Descomposición de LOTES1 en las relaciones 3FN LOTES1A y LOTES1B. (d) Resumen de la normalización de LOTES.

TASA_FISCAL que viola 2FN y colocándolo junto con NOMBRE_MUNIC (el miembro izquierdo de df3 que causa la dependencia parcial) en otra relación LOTES2. Tanto LOTES1 como LOTES2 están en 2FN. Adviértase que df4 no viola 2FN y persiste en LOTES1.

12.4.2 *Definición general de tercera forma normal (3FN)*

Un esquema de relación R está en 3FN si, siempre que una dependencia funcional $X \rightarrow A$ se cumple en R, o bien (a) X es una superclave de R, o (b) A es un atributo primo de R. De acuerdo con esta definición, LOTES2 (Fig. 12.11(b)) está en 3FN. Sin embargo, df4 de LOTES1 viola 3FN porque ÁREA no es una superclave de LOTES1 y PRECIO no es un atributo primo. Para normalizar LOTES1 a 3FN, la descomponemos en los esquemas de relación LOTES1A y LOTES1B, como en la figura 12.11(c). Construimos LOTES1A eliminando de LOTES1 el atributo PRECIO que viola 3FN y colocándolo junto con ÁREA (el miembro izquierdo de df4 que causa la dependencia transitiva) en otra relación LOTES1B. Tanto LOTES1A como LOTES1B están en 3FN. Vale la pena destacar dos puntos acerca de la definición general de 3FN:

- Esta definición puede aplicarse *directamente* para comprobar si un esquema de relación está en 3FN; *no* tiene que pasar primero por 2FN . Si aplicamos la definición 3FN a LOTES con las dependencias df1 a df4, veremos que *tanto* df3 *como* df4 violan 3FN. Así pues, podríamos descomponer LOTES en LOTES1A, LOTES1B y LOTES2 directamente.
- LOTES1 viola 3FN porque PRECIO depende transitivamente de cada una de las claves candidatas de LOTES1 a través del atributo no primo ÁREA.

12.4.3 *Interpretación de la definición general de 3FN*

Un esquema de relación R viola la definición general de 3FN si una dependencia funcional $X \rightarrow A$ válida en R viola ambas condiciones, (a) y (b), de 3FN. La violación de (b) implica que A es un atributo no primo. La violación de (a) implica que X no es un superconjunto de ninguna clave de R; por tanto, X podría ser no primo o podría ser un subconjunto propio de una clave de R. Si X no es primo, por lo regular tenemos una dependencia transitiva que viola 3FN, y si X es un subconjunto propio de una clave de R, tenemos una dependencia parcial que viola 3FN (y también 2FN). Por tanto, podemos expresar una **definición general alternativa de 3FN** como sigue: Un esquema de relación R está en 3FN si todo atributo no primo de R es:

- dependiente funcionalmente de manera total de toda clave de R, y
- dependiente de manera no transitiva de toda clave de R.

12.5 Forma normal de Boyce-Codd (FNBC)*

La forma normal de Boyce-Codd es más estricta que la 3FN, lo que significa que toda relación que esté en FNBC también está en 3FN; sin embargo, una relación en 3FN *no está necesariamente* en FNBC. Intuitivamente, podemos ver por qué es necesaria una forma normal más estricta que la 3FN si volvemos al esquema de relación LOTES de la figura 12.11(a) con sus cuatro dependencias funcionales df1 a df4. Suponga que tenemos miles de lotes en la relación pero que dichos lotes pertenecen a sólo dos municipios: Malinalco y Libertad. Suponga también que los tamaños de los lotes en el municipio Malinalco son de sólo 0.5, 0.6, 0.7, 0.8, 0.9 y 1.0 hectáreas, mientras que los tamaños de los lotes en el municipio

Libertad están restringidos a 1.1, 1.2, ..., 1.9 y 2.0 hectáreas. En una situación así tendríamos la dependencia funcional adicional df5: ÁREA → NOMBRE_MUNIC. Si añadimos ésta a las demás dependencias, el esquema de relación LOTES1A seguirá estando en 3FN porque NOMBRE_MUNIC es un atributo primo.

La interrelación de área y municipio representada por df5 puede representarse con 16 tuplas en una relación aparte R(ÁREA, NOMBRE_MUNIC), ya que sólo hay 16 posibles valores de ÁREA. Esta representación reduce la redundancia de repetir la misma información en las miles de tuplas LOTES1A. FNBC es una *forma normal más estricta* que prohibiría LOTES1A y sugeriría que habría que descomponerla.

Esta definición de Boyce-Codd difiere un poco de la definición de 3FN. Un esquema de relación R está en FNBC si, siempre que una dependencia funcional $X \rightarrow A$ es válida en R, entonces X es una superclave de R. La única diferencia entre FNBC y 3FN es que la condición (b) de 3FN, que permite que A sea primo si X no es una superclave, está ausente en FNBC.

En nuestro ejemplo, df5 viola FNBC en LOTES1A porque ÁREA no es una superclave de LOTES1A. Observe que df5 satisface 3FN en LOTES1A porque NOMBRE_MUNIC es un atributo primo (condición (b)), pero esta condición no existe en la definición de FNBC. Podemos descomponer LOTES1A en las dos relaciones FNBC LOTES1AX y LOTES1AY, que aparecen en la figura 12.12(a).

En la práctica, casi todos los esquemas de relación que están en 3FN también están en FNBC. Sólo si existe una dependencia $X \rightarrow A$ en un esquema de relación R, y X no es una superclave y A es un atributo primo, R estará en 3FN pero no en FNBC. El esquema de relación R que se aprecia en la figura 12.12(b) ilustra el caso general de una relación así. Es mejor tener los esquemas de relación en FNBC; si esto no es posible, bastará con que estén en 3FN. Sin embargo, ni 2FN ni 1FN se consideran buenos diseños de esquemas de relación. Estas formas normales se desarrollaron históricamente como escalones para llegar a 3FN y FNBC.

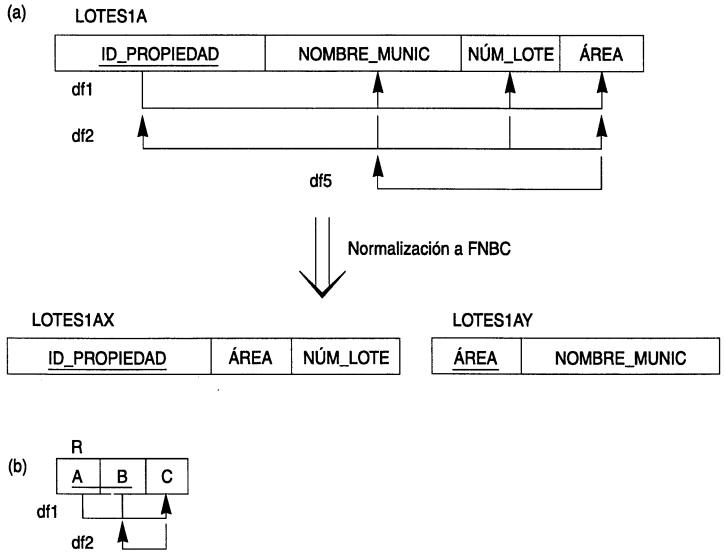


Figura 12.12 FNBC. (a) Normalización a FNBC: la dependencia df2 se "pierde" en la descomposición. (b) Relación R que está en 3FN pero no en FNBC.

12.6 Resumen

En este capítulo estudiamos desde una perspectiva intuitiva varios errores que pueden cometerse al diseñar bases de datos relacionales, y luego presentamos algunos conceptos formales básicos que son decisivos en el diseño de una base de datos relacional. En el capítulo 13 continuaremos con los temas tratados en este capítulo y examinaremos conceptos más avanzados de la teoría del diseño relacional.

En la sección 12.1 analizamos de manera informal algunas de las medidas que indican si un esquema de relación es "bueno" o "malo", y explicamos ciertas pautas informales para lograr un buen diseño. Examinamos los problemas de las anomalías de actualización que se presentan cuando hay redundancia en las relaciones. Otras medidas informales de los buenos esquemas de relación son una semántica de atributos simple y clara y pocos nulos en las relaciones correspondientes a los esquemas.

En la sección 12.1.4 tratamos el problema de las tuplas espurias. Este problema se formalizará y resolverá en el capítulo 13, donde presentaremos algoritmos de descomposición para el diseño de bases de datos relacionales a partir de dependencias funcionales. Ahí estudiaremos los conceptos de "reunión sin pérdida" y "conservación de dependencias" que en algunos de estos algoritmos son obligatorios. La propiedad de reunión sin pérdida garantiza que no habrá tuplas espurias. En el capítulo 13 veremos, entre otros temas, las dependencias multivaluadas, las dependencias de reunión y las formas normales adicionales que tienen en cuenta estas dependencias.

En la sección 12.2 examinamos el concepto de dependencia funcional y analizamos algunas de sus propiedades. Las dependencias funcionales son restricciones fundamentales que se especifican entre los atributos de un esquema de relación. Mostramos cómo se pueden inferir las dependencias funcionales a partir de un conjunto de dependencias y cómo verificar si dos conjuntos de dependencias funcionales son equivalentes.

En la sección 12.3 usamos el concepto de dependencia funcional para definir las formas normales con base en las claves primarias. En la sección 12.4 proporcionamos definiciones más generales de la segunda forma normal (2FN) y de la tercera forma normal (3FN), teniendo en cuenta todas las claves candidatas de una relación. Por último, presentamos la forma normal de Boyce-Codd (FNBC) en la sección 12.5 y examinamos en qué se diferencia de 3FN. Ilustramos con ejemplos la forma de usar estas formas normales para descomponer una relación no normalizada hasta obtener un conjunto de relaciones en 3FN o FNBC.

Preguntas de repaso

12.1. Analice la semántica de los atributos como medida informal de la "bondad" de un esquema de relación.

12.2. Explique las anomalías de inserción, eliminación y modificación. ¿Por qué se consideran indeseables?

12.3. ¿Por qué no se considera bueno tener muchos nulos en una relación?

12.4. Analice el problema de las tuplas espurias y la forma de evitarlo.

12.5. Comente las pautas informales para diseñar esquemas de relación.

12.6. ¿Qué es una dependencia funcional? ¿Quién especifica las dependencias funcionales que han de mantenerse entre los atributos de un esquema de relación?

12.7. ¿Por qué no podemos deducir una dependencia funcional a partir de un ejemplar de relación específico?

12.8. ¿A qué nos referimos cuando decimos que las reglas de inferencia de Armstrong son correctas y completas?

12.9. ¿Qué es la cerradura de un conjunto de dependencias funcionales?

12.10. ¿Cuándo son equivalentes dos conjuntos de dependencias funcionales? ¿Cómo podemos determinar su equivalencia?

12.11. ¿Qué es un conjunto mínimo de dependencias funcionales? ¿Tiene todo conjunto de dependencias un conjunto equivalente mínimo?

12.12. Defina las formas normales primera, segunda y tercera cuando sólo se consideran las claves primarias. ¿Qué diferencias hay entre las definiciones generales de 2FN y 3FN, que consideran todas las claves de una relación, y las que consideran sólo las claves primarias?

12.13. ¿Por qué en general se considera buena una relación que está en 3FN?

12.14. Defina la forma normal de Boyce-Codd. ¿Qué diferencias presenta respecto a 3FN?

12.15. ¿Cómo se desarrollaron históricamente las formas normales?

Ejercicios

12.16. Suponga que tenemos los siguientes requerimientos para una base de datos universitaria con que se manejan las boletas de notas de los estudiantes:

a. Para cada estudiante, la universidad mantiene su nombre (NOMBREE), su número (NÚMEST), su número de seguro social (NSSE), su dirección y teléfono actuales (DIRACEST y TELACEST), su dirección y teléfono permanentes (DIRPEREST y TELPEREST), su fecha de nacimiento (FECHAN), su sexo (SEXO), su grado (GRADO) (primero, segundo, ..., graduado), su departamento de carrera (CÓDDEPCARR), su departamento de especialización (CÓDDEPESP) (si lo tiene) y su programa de grado (PROG) (B.A., B.C., ..., DOC). Tanto NSS como NÚMEST tienen valores únicos para cada estudiante.

b. Cada departamento se describe mediante un nombre (NOMDEPTO), un código (CÓDDEPTO), un número de oficina (OFICDEPTO), un teléfono de oficina (TELDEPTO) y un colegio (COLEGIODEPTO). Tanto el nombre como el código tienen valores únicos para cada departamento.

c. Cada curso tiene un nombre (NOMCURSO), una descripción (DESCCUR), un código numérico (NÚMCURSO), un número de horas por semestre (CRÉDITO) un nivel (NIVEL) y un departamento que lo ofrece (DEPCURSO). El valor del código numérico es único para cada curso.

d. Cada sección (curso impartido) tiene un profesor (NOMBREPROF), un semestre (SEMESTRE), un año (AÑO), un curso (CURSOSEC) y un número de sección (NÚMSEC). El número de sección distingue las diferentes secciones (grupos) del mismo curso impartidas durante el mismo semestre/año; sus valores son 1, 2, 3, ...; hasta el número total de secciones impartidas durante cada semestre.

e. Una boleta se refiere a un estudiante (NSSE), a una sección (SECCIÓN) y a una nota (NOTAS) determinados.

Diseñe un esquema de base de datos relacional para esta aplicación. Primero indique todas las dependencias funcionales que deben cumplirse entre los atributos. Luego diseñe esquemas de relación para la base de datos que estén en 3FN o FNBC. Especifique los atributos clave de cada relación. Señale cualesquier requerimientos que no se hayan especificado y haga las suposiciones apropiadas para completar dichas especificaciones.

12.17. Demuestre o refute las siguientes reglas de inferencia para las dependencias funcionales. Puede hacerse la demostración con un argumento de demostración o con las reglas de inferencia RI1, RI2 y RI3. La refutación deberá efectuarse citando un ejemplar de relación que satisfaga las condiciones y dependencias funcionales del miembro izquierdo de la regla de inferencia, pero que no satisfaga las dependencias del miembro derecho.

a. $\{W \rightarrow Y, X \rightarrow Z\} \models \{WX \rightarrow Y\}$.
b. $\{X \rightarrow Y\}$ y $Z \subseteq Y \models \{X \rightarrow Z\}$.
c. $\{X \rightarrow Y, X \rightarrow W, WY \rightarrow Z\} \models \{X \rightarrow Z\}$.
d. $\{XY \rightarrow Z, Y \rightarrow W\} \models \{XW \rightarrow Z\}$.
e. $\{X \rightarrow Z, Y \rightarrow Z\} \models \{X \rightarrow Y\}$.
f. $\{X \rightarrow Y, XY \rightarrow Z\} \models \{X \rightarrow Z\}$.
g. $\{X \rightarrow Y, Z \rightarrow W\} \models \{XZ \rightarrow YW\}$.
h. $\{XY \rightarrow Z, Z \rightarrow X\} \models \{Z \rightarrow Y\}$.
i. $\{X \rightarrow Y, Y \rightarrow Z\} \models \{X \rightarrow YZ\}$.
j. $\{XY \rightarrow Z, Z \rightarrow W\} \models \{X \rightarrow W\}$.

12.18. ¿Por qué son importantes las tres reglas de inferencia RI1, RI2 y RI3 (reglas de inferencia de Armstrong)?

12.19. Considere los siguientes dos conjuntos de dependencias funcionales: $F = \{A \rightarrow C, AC \rightarrow D, E \rightarrow AD, E \rightarrow H\}$ y $G = \{A \rightarrow CD, E \rightarrow AH\}$. Compruebe si son equivalentes.

12.20. Considere el esquema de relación EMP_DEPTO de la figura 12.3(a) y el siguiente conjunto G de dependencias funcionales en EMP_DEPTO: G = {NSS→ {NOMBREE, FECHAN, DIRECCIÓN, NÚMEROD}, NÚMEROD→ {NOMBRED, NSSGTED}}. Calcule las cerraduras $\{NSS\}^+$ y $\{NÚMEROD\}^+$ con respecto a G.

12.21. ¿Es mínimo el conjunto de dependencias funcionales G del ejercicio 12.20? Si no, trate de encontrar un conjunto mínimo de dependencias funcionales que sea equivalente a G. Demuestre que su conjunto es equivalente a G.

12.22. ¿Por qué se consideran malas en un esquema relacional las dependencias transitivas y las dependencias parciales?

12.23. ¿Qué anomalías de actualización ocurren en las relaciones EMP_PROY y EMP_DEPTO de las figuras 12.3 y 12.4?

12.24. ¿En qué forma normal está el esquema de relación LOTES de la figura 12.11(a) con respecto a las interpretaciones restrictivas de forma normal que tienen en cuenta *sólo la clave primaria*? ¿Estaría en la misma forma normal si se usaran las definiciones generales de forma normal?

12.25. ¿Por qué se considera que FNBC es mejor que 3FN?

12.26. Demuestre que cualquier esquema de relación con dos atributos está en FNBC.

12.27. ¿Por qué aparecen tuplas espurias en el resultado de reunir las relaciones EMP_PROY1 y LUGARES_EMPS de la figura 12.5? (El resultado aparece en la figura 12.6.)

12.28. Considere la relación universal $R = \{A, B, C, D, E, F, G, H, I, J\}$ y el conjunto de dependencias funcionales $F = \{\{A, B\} \rightarrow \{C\}, \{A\} \rightarrow \{D, E\}, \{B\} \rightarrow \{F\}, \{F\} \rightarrow \{G, H\}, \{D\} \rightarrow \{I, J\}\}$. ¿Cuál es la clave de R? Descomponga R en relaciones 2FN y luego 3FN.

12.29. Repita el ejercicio 12.28 con un conjunto de dependencias funcionales distinto, a saber: $G = \{\{A, B\} \rightarrow \{C\}, \{B, D\} \rightarrow \{E, F\}, \{A, D\} \rightarrow \{G, H\}, \{A\} \rightarrow \{I\}, \{H\} \rightarrow \{J\}\}$.

12.30. Considere la relación R que contiene atributos relativos a los horarios de cursos y secciones en una universidad: R = {NúmCurso, NúmSec, DeptoOfrece, HorasCrédito, NivelCurso, NSSProfesor, Semestre, Año, Días_Horas, NúmAula, NúmDeEstudiantes}. Suponga que son válidas las siguientes dependencias funcionales en R:

{NúmCurso} → {DeptoOfrece, HorasCrédito, NivelCurso}

{NúmCurso, NúmSec, Semestre, Año} →

{Días_Horas, NúmAula, NúmDeEstudiantes, NSSProfesor}

{NúmAula, Días_Horas, Semestre, Año} →

{NSSProfesor, NúmCurso, NúmSec}

Trate de determinar cuáles conjuntos de atributos forman claves de R. ¿Cómo normalizaría esta relación?

Bibliografía selecta

La exposición original sobre las dependencias funcionales fue de Codd (1970). Las definiciones originales de la primera, segunda y tercera formas normales también se deben a Codd (1972*a*), y ahí mismo puede encontrarse una explicación de las anomalías de actualización. La forma normal de Boyce-Codd se definió en Codd (1974). La definición alternativa de tercera forma normal aparece en Ullman(1988), lo mismo que la definición de FNBC que aquí dimos. Los textos de Ullman (1988) y Maier (1983) contienen muchos de los teoremas y demostraciones relativos a las dependencias funcionales.

Armstrong (1974) demuestra la corrección y compleción de las reglas de inferencia RI1, RI2 y RI3. En el capítulo 13 se darán referencias adicionales sobre la teoría del diseño relacional.

CAPÍTULO 13

Algoritmos de diseño de bases de datos relacionales y dependencias adicionales

Hay dos técnicas principales para diseñar esquemas de bases de datos relacionales. La primera implica diseñar un esquema conceptual en un modelo de datos de alto nivel, como el modelo ER, y luego transformar el esquema conceptual a un conjunto de relaciones empleando un procedimiento de transformación como el que vimos en la sección 6.8. Esto puede denominarse **diseño descendente**. En esta técnica podemos aplicar informalmente los principios de normalización que estudiamos en el capítulo 12, como evitar las dependencias transitivas y parciales, *tanto* al diseño del esquema conceptual *como* a las relaciones que resultan del procedimiento de transformación. El segundo enfoque, más purista, implica contemplar el diseño de esquemas de bases de datos relacionales estrictamente en términos de las dependencias funcionales y de otros tipos especificadas para los atributos de la base de datos. Esto suele denominarse **síntesis relacional**, porque los esquemas de relación en 3FN y FNBC se *sintetizan* agrupando los atributos apropiados. Cada esquema de relación individual debe representar una agrupación lógicamente coherente de atributos y debe poseer las características de "bondad" asociadas a la normalización.

En el capítulo 12 estudiamos algunas medidas de bondad para esquemas de relación individuales basados en sus claves y sus dependencias funcionales; a saber, que estén en FNBC o —si esto no es posible— en 3FN. Durante el proceso de normalización, **descomponemos** una relación que no está en una cierta forma normal produciendo múltiples esquemas de relación hasta lograr un diseño final con relaciones en la forma normal deseada. Un caso extremo de este proceso de diseño se denomina **descomposición estricta**, en la cual comenzamos por sintetizar un esquema de relación gigante, llamado **relación universal**, que incluye todos los atributos de la base de datos. A continuación realizamos repetidamente una descomposición hasta que ya no sea factible o deseable seguir.

En este capítulo primero presentaremos varios algoritmos basados en dependencias funcionales que pueden servir para el diseño de bases de datos relacionales. En la sección 13.1 examinaremos los conceptos de conservación de las dependencias y reuniones sin pérdidas (o no aditivas), con los que los algoritmos de diseño efectúan las descomposiciones deseables. Demostraremos que las formas normales son *insuficientes por sí solas* como criterios para diseñar un buen esquema de base de datos relacional.

Después estudiaremos otros tipos de dependencias con respecto a los datos y algunas de las formas normales que producen. Estas dependencias especifican restricciones que *no pueden* expresarse mediante dependencias funcionales. Trataremos las dependencias multivaluadas, las dependencias de reunión, las dependencias de inclusión y las dependencias de patrón. También definiremos la cuarta forma normal (4FN), la quinta forma normal (5FN) y, someramente, la forma normal de dominio-clave (FNDC: *domain-key normal form*).

Es posible pasar por alto algunas de las siguientes secciones, o todas: 13.3, 13.4, 13.5.

13.1 Algoritmos para el diseño de esquemas de bases de datos relacionales

En la sección 13.1.1 daremos ejemplos con los que se verá que comprobar que una relación *individual* está en una forma normal superior no garantiza un buen diseño; más bien, un *conjunto de relaciones* que juntas constituyen el esquema de base de datos relacional deben poseer ciertas propiedades adicionales para asegurar un buen diseño. En la sección 13.1.2 trataremos una de esas propiedades, la de conservación de las dependencias. En la sección 13.1.3 estudiaremos otra de esas propiedades, la de reunión sin pérdidas o no aditiva. Presentaremos algoritmos de descomposición que garanticen estas propiedades (que son conceptos formales) y que además aseguren que las relaciones individuales estén correctamente normalizadas. En la sección 13.1.4 veremos los problemas asociados a los valores nulos, y en la sección 13.1.5 resumiremos los algoritmos de diseño y sus propiedades.

13.1.1 *Descomposición de relaciones e insuficiencia de las formas normales*

Los algoritmos de diseño de bases de datos relacionales que presentamos aquí parten de un solo **esquema de relación universal** $R = \{A_1, A_2, ..., A_n\}$ que contiene *todos* los atributos de la base de datos. Hacemos implícitamente la suposición de relación universal, que nos dice que todos los nombres de atributos son únicos. El conjunto F de dependencias funcionales que deben cumplir los atributos de R lo especifican los diseñadores de la base de datos y está disponible para los algoritmos de diseño. Con las dependencias funcionales, los algoritmos **descomponen** el esquema de relación universal R en un conjunto de esquemas de relación $D = \{R_1, R_2, ..., R_m\}$, que se convertirá en el esquema de la base de datos relacional; D es una **descomposición** de R.

Debemos asegurarnos de que todos los atributos de R aparezcan en por lo menos un esquema de relación R_i de la descomposición, de modo que no se "pierdan" atributos. En términos formales:

$$\bigcup_{i=1}^{m} R_i = R$$

Ésta es la condición de **conservación de atributos** de una descomposición.

Otro objetivo es lograr que cada relación individual R_i de la descomposición D esté en FNBC (o 3FN). Sin embargo, esta condición no basta por sí sola para garantizar un buen diseño de base de datos. Debemos considerar la descomposición como un todo, además de examinar las relaciones individuales. Para ilustrar este punto, consideremos la relación LUGARES_EMPS de la figura 12.5, que está en 3FN y también en FNBC. De hecho, cualquier esquema de relación que sólo tenga dos atributos estará automáticamente en FNBC (Ejercicio 12.26). Aunque LUGARES_EMPS está en FNBC, producirá tuplas espurias cuando se le reúna con EMP_PROY1 (que no está en FNBC) (Fig. 12.6). Por esto, LUGARES_EMPS representa un esquema de relación notablemente malo debido a lo rebuscado de su semántica según la cual LUGARP da la ubicación de *uno de los proyectos* en los que un empleado trabaja. Si reunimos LUGARES_EMPS con PROYECTO de la figura 12.2 (que está en FNBC) también tendremos tuplas espurias. Necesitamos otros criterios que, aunados a las condiciones de 3FN o FNBC, eviten tales diseños deficientes. En las siguientes tres subsecciones examinaremos las condiciones adicionales que debe satisfacer globalmente una descomposición D.

13.1.2 *Descomposición y conservación de las dependencias*

Sería útil que toda dependencia funcional $X \rightarrow Y$ especificada en F apareciera directamente en uno de los esquemas de relación R_i de la descomposición D o bien que pudiera inferirse de las dependencias que aparecen en alguna R_i. Informalmente, ésta es la condición de conservación de las dependencias. Queremos conservarlas porque cada dependencia en F representa una restricción sobre la base de datos. Si alguna de las dependencias no está representada en alguna relación individual R_i de la descomposición, no podremos imponer esta restricción con sólo examinar una relación individual; en vez de ello, tendremos que reunir dos o más relaciones de la descomposición y luego verificar que la dependencia funcional se cumpla en el resultado de la operación de reunión. Esto es a todas luces un procedimiento ineficiente que no servirá en un sistema práctico.

No es necesario que las dependencias exactas especificadas en F aparezcan en relaciones individuales de la descomposición D. Basta con que la unión de las dependencias que se cumplen en las relaciones individuales de D sea equivalente a F. Ahora definiremos estos conceptos en términos más formales. Primero necesitamos una definición preliminar: dado un conjunto de dependencias F sobre R, la **proyección** de F sobre R_i, denotada por $\pi_F(R_i)$ donde R_i es un subconjunto de R, es el conjunto de dependencias $X \rightarrow Y$ en F^+ tal que los atributos en $X \cup Y$ estén todos contenidos en R_i. Así pues, la proyección de F sobre cada esquema de relación R_i de la descomposición D es el conjunto de dependencias funcionales en F^+, la cerradura de F, tales que todos sus atributos de miembro izquierdo y miembro derecho estén en R_i. Decimos que una descomposición $D = \{R_1, R_2, ..., R_m\}$ de R es **conservadora de las dependencias** respecto a F si la unión de las proyecciones de F sobre cada R_i de D es equivalente a F; esto es,

$$((\pi_F(R_1)) \cup \cdots \cup (\pi_F(R_m)))^+ = F^+$$

Si una descomposición no conserva las dependencias, alguna dependencia se **perderá** en la descomposición. Como ya mencionamos, para comlprobar si se cumple o no una dependencia perdida deberemos obtener la REUNIÓN de varias relaciones de la descomposición para obtener una relación que incluya todos los atributos de los miembros izquierdo y derecho de la dependencia perdida, y luego comprobar que la dependencia sea válida en el resultado de la REUNIÓN, opción que no resulta práctica.

En la figura 12.12(a) se muestra un ejemplo de descomposición que no conserva las dependencias, pues ahí se pierde la dependencia funcional df2 cuando LOTES1A se descompone en {LOTES1AX, LOTES1AY}. Las descomposiciones de la figura 12.11, en cambio, sí conservan las dependencias. Siempre es posible encontrar una descomposición D conservadora de las dependencias respecto a F tal que toda relación R_i en D esté en 3FN. El algoritmo 13.1 crea una descomposición de esta índole. Este algoritmo garantiza sólo la propiedad de conservación de las dependencias; *no* garantiza la propiedad de reunión sin pérdidas que analizaremos en la siguiente sección. El primer paso del algoritmo 13.1 es encontrar una cobertura mínima G para F. Un algoritmo para llevar a cabo este paso se bosqueja como algoritmo 13.1*a* en seguida. Recordemos, de la sección 12.2.4, que una cobertura mínima G de F es un *conjunto mínimo* de dependencias funcionales equivalente a F. El algoritmo 13.1*a* se basa en las tres condiciones que debe satisfacer un conjunto mínimo de dependencias (véase la Sec. 12.2.4). El paso 2 del algoritmo se encarga de que toda dependencia funcional en G tenga un solo atributo como miembro derecho. El paso 3 garantiza que no quedarán dependencias funcionales redundantes en G. El paso 4 elimina cualesquier atributos redundantes del miembro izquierdo de una dependencia funcional.

ALGORITMO 13.1 Descomposición en esquemas de relación 3FN conservando las dependencias

1. encontrar una cobertura mínima G para F;
2. para cada miembro izquierdo X de una dependencia funcional que aparezca en G
   crear un esquema de relación $\{X$ UNIÓN A_1 UNIÓN A_2 ... UNIÓN $A_m\}$ en D, donde $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_m$ sean las únicas dependencias en G con X como miembro izquierdo;
3. colocar cualesquier atributos restantes (no colocados) en un solo esquema de relación para asegurar la propiedad de conservación de las dependencias;

ALGORITMO 13.1*a* Encontrar una cobertura mínima G para F

1. hacer $G := F$;
2. reemplazar cada dependencia funcional $X \rightarrow A_1, A_2, ..., A_n$ en G por las n dependencias funcionales $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_n$;
3. para cada dependencia funcional $X \rightarrow A$ en G
   {calcular X^+ respecto al conjunto de dependencias $(G - (X \rightarrow A))$;
   si X^+ contiene a A, eliminar $X \rightarrow A$ de G };
4. para cada dependencia funcional restante $X \rightarrow A$ en G
   para cada atributo B que sea un elemento de X
   {calcular $(X - B)^+$ respecto al conjunto de dependencias funcionales $((G - (X \rightarrow A))$ UNIÓN $((X - B) \rightarrow A))$;
   si $(X - B)^+$ contiene a A, reemplazar $X \rightarrow A$ por $(X - B) \rightarrow A$ en G};

No haremos una demostración formal, pero es posible comprobar que todo esquema de relación creado por el algoritmo 13.1 está en 3FN. La demostración se basa en el hecho de que G es una cobertura mínima, de modo que las dependencias en G satisfacen las propiedades mencionadas en la sección 12.2.4. Es obvio que el algoritmo conserva todas las dependencias en G porque cada una de las dependencias aparece en una de las relaciones R_i de la descomposición D. Puesto que G es una cobertura mínima de F, es equivalente a F, y todas las dependencias en F se conservan directamente en la descomposición o bien pueden derivarse de las que se cumplen en las relaciones resultantes. El algoritmo 13.1 se denomina algoritmo de **síntesis relacional** porque cada uno de los esquemas de relación R_i en la descomposición se "sintetiza" a partir de un conjunto de dependencias en G con el mismo miembro izquierdo.

13.1.3 *Descomposición y reuniones sin pérdidas (no aditivas)*

Otra propiedad que debe poseer una descomposición es la de reunión sin pérdidas o no aditiva, la cual garantiza que no se generarán tuplas espurias cuando se aplique una operación de REUNIÓN NATURAL a las relaciones de la descomposición. Ya ilustramos este problema en la sección 12.1.4 con el ejemplo de las figuras 12.5 y 12.6. Como ésta es una propiedad de una descomposición de esquemas de relación, la condición de ausencia de tuplas espurias deberá cumplirse en *todos los ejemplares de relación permitidos*; esto es, todos los ejemplares de relación que satisfagan las dependencias funcionales especificadas sobre los esquemas. Por ello, la propiedad de reunión sin pérdidas siempre se define con respecto a un conjunto específico de dependencias F. En términos formales, una descomposición $D = \{R_1, R_2, ..., R_m\}$ de R tiene la propiedad de **reunión sin pérdidas (no aditiva)** respecto al conjunto de dependencias F sobre R si, por cada estado de relación r de R que satisfaga F, se cumple lo siguiente:

$$* (\pi_{<R1>}(r), ..., \pi_{<Rm>}(r)) = r$$

El término *sin pérdidas* se refiere a la pérdida de información, no a la pérdida de tuplas. Si una descomposición no posee la propiedad de reunión sin pérdidas, es posible que tengamos tuplas espurias adicionales después de aplicar las operaciones PROYECTAR (π) y REUNIÓN NATURAL (*); estas tuplas adicionales representan información errónea. Preferimos el término **reunión no aditiva** porque describe la situación con mayor exactitud; si la propiedad es válida en una descomposición, estaremos seguros de que no se *añadirán* tuplas espurias con información errónea al resultado de aplicar las operaciones PROYECTAR y REUNIÓN NATURAL.

Es obvio que la descomposición de EMP_PROY en LUGARES_EMPS y EMP_PROY1 en la figura 12.5 no posee la propiedad de reunión sin pérdidas. En general, querremos poder verificar si una descomposición D dada posee la propiedad de reunión sin pérdidas respecto a un conjunto de dependencias funcionales F. Podemos emplear el algoritmo 13.2 para efectuar esta comprobación.

El algoritmo 13.2 crea un ejemplar de relación r en la matriz S que satisface todas las dependencias funcionales en F. Al final del ciclo de aplicación de dependencias funcionales, cualesquier dos filas de S —que representan dos tuplas de r— que coincidan en sus valores para los atributos del miembro izquierdo X de una dependencia funcional $X \rightarrow Y$ en F, también coincidirán en sus valores para los atributos del miembro derecho Y. Puede mostrarse que, si cualquier fila de S termina únicamente con símbolos "a" al final del algoritmo, la descomposición poseerá la propiedad de reunión sin pérdidas respecto a F. Si, en cambio, ninguna fila termina sólo con símbolos "a", el ejemplar de relación r representado por S al final del algoritmo será un ejemplo de ejemplar de relación r de R que satisfará las dependencias en F pero que no poseerá la propiedad de reunión sin pérdidas. Esta relación sirve

como contraejemplo, así que la descomposición D no tiene la propiedad de reunión sin pérdidas en este caso. Adviértase que los símbolos "a" y "b" no tienen ningún significado especial al final del algoritmo.

La figura 13.1(a) muestra cómo aplicamos el algoritmo 13.2 a la descomposición del esquema de relación EMP_PROY de la figura 12.3(b) para dar los dos esquemas de relación EMP_PROY1 y LUGARES_EMPS de la figura 12.5(a). El algoritmo no puede cambiar ningún símbolo "b" a "a", así que la descomposición no posee la propiedad de reunión sin pérdidas.

La figura 13.1(b) muestra otra descomposición de EMP_PROY que posee la propiedad de reunión sin pérdidas, y la figura 13.1(c) ilustra cómo aplicamos el algoritmo a esa descomposición. Tan pronto como una fila conste únicamente de símbolos "a", sabremos que la descomposición posee la propiedad de reunión sin pérdidas, y podremos dejar de aplicar las dependencias funcionales a la matriz S.

ALGORITMO 13.2 Comprobación de la propiedad de reunión sin pérdidas

1. crear una matriz S con una fila i por cada relación R_i en la descomposición D, y una columna j por cada atributo A_j en R;
2. hacer $S(i,j):=b_{ij}$ para todas las entradas de la matriz;
   (* cada b_{ij} es un símbolo distinto asociado a los índices (i,j) *)
3. para cada fila i que represente el esquema de relación R_i
   para cada columna j que represente el atributo A_j
   si R_i incluye el atributo A_j
   entonces hacer $S(i,j):= a_j$;
   (* cada a_j es un símbolo distinto asociado al índice (j) *)
4. repetir lo que sigue hasta que una ejecución no modifique S
   para cada dependencia funcional $X \rightarrow Y$ en F
   para todas las filas en S *que tienen los mismos símbolos* en las columnas correspondientes a los atributos en X
   hacer que los símbolos de cada columna que correspondan a un atributo en Y sean iguales en todas estas filas como sigue: si cualquiera de las filas tiene un símbolo "a" en la columna, asignar el *mismo* símbolo "a" a las otras filas en esa columna; si no hay ningún símbolo "a" para el atributo en ninguna de las filas, escoger uno de los símbolos "b" que aparecen en una de las filas del atributo y asignar ese símbolo "b" a las otras filas en esa columna;
5. si una fila consta exclusivamente de símbolos "a", la descomposición posee la propiedad de reunión sin pérdidas; en caso contrario, no la posee;

Ahora podemos comprobar si una descomposición D específica obedece la propiedad de reunión sin pérdidas respecto a un conjunto de dependencias funcionales F. La siguiente pregunta es si hay algún algoritmo para descomponer un esquema de relación $R = \{A_1, A_2, ..., A_n\}$ en una descomposición $D = \{R_1, R_2, ..., R_m\}$, tal que cada R_i esté en FNBC y la descomposición D tenga la propiedad de reunión sin pérdidas respecto a F. La respuesta es afirmativa; existe un algoritmo así, pero antes de darlo necesitamos presentar algunas propiedades de las descomposiciones con reunión sin pérdidas.

(a) R={NSS, NOMBREE, NÚMEROP, NOMBREPR, LUGARP, HORAS} D={R1, R2}
R1=LUGARES_EMPS={NOMBREE, LUGARP}
R2=EMP_PROY1={NSS, NÚMEROP, HORAS, NOMBREPR, LUGARP}

F={NSS→ NOMBREE; NÚMEROP→ {NOMBREPR, LUGARP}; {NSS, NÚMEROP}→ HORAS}

	NSS	NOMBREE	NÚMEROP	NOMBREPR	LUGARP	HORAS
R1	b_{11}	a_2	b_{13}	b_{14}	a_5	b_{16}
R2	a_1	b_{22}	a_3	a_4	a_5	a_6

(la matriz no cambia después de aplicar dependencias funcionales)

(b)

EMP	
NSS	NOMBREE

PROYECTO		
NÚMEROP	NOMBREPR	LUGARP

TRABAJA_EN		
NSS	NÚMEROP	HORAS

(c) R={NSS, NOMBREE, NÚMEROP, NOMBREPR, LUGARP, HORAS} D={R1, R2, R3}
R1=EMP={NSS, NOMBREE}
R2=PROYECTO={NÚMEROP, NOMBREPR, LUGARP}
R3=TRABAJA_EN={NSS, NÚMEROP, HORAS}

F={NSS →NOMBREE; NÚMEROP →{NOMBREPR, LUGARP}; {NSS, NÚMEROP} →HORAS}

	NSS	NOMBREE	NÚMEROP	NOMBREPR	LUGARP	HORAS
R1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

(matriz original S al principio del algoritmo)

	NSS	NOMBREE	NÚMEROP	NOMBREPR	LUGARP	HORAS
R1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R3	a_1	~~b_{32}~~ a_2	a_3	~~b_{34}~~ a_4	~~b_{35}~~ a_5	a_6

(la matriz S después de aplicar las dos primeras dependencias funcionales - la última fila sólo tiene símbolos "a", así que nos detenemos)

Figura 13.1 El algoritmo de comprobación de la reunión sin pérdidas. (a) Aplicación de este algoritmo a la descomposición de EMP_PROY en EMP_PROY1 y LUGARES_EMPS. (b) Otra descomposición de EMP_PROY. (c) Aplicación del algoritmo a la descomposición de la figura 13.1(b).

PROPIEDAD RSP1

Una descomposición $D = \{R_1, R_2\}$ de R tiene la propiedad de reunión sin pérdidas respecto a un conjunto de dependencias funcionales F sobre R si y sólo si:

- la DF $((R_1 \cap R_2) \rightarrow (R_1 - R_2))$ está en F^+, o bien
- la DF $((R_1 \cap R_2) \rightarrow (R_2 - R_1))$ está en F^+.

Observe que la propiedad RSP1 constituye una prueba más sencilla de la propiedad de reunión sin pérdidas que el algoritmo 13.2. Sin embargo, RSP1 *sólo* es aplicable a descomposiciones en *dos esquemas de relación*. Recomendamos al lector verificar que esta propiedad se cumpla en nuestros ejemplos de normalización informal sucesiva de las secciones 12.3 y 12.4.

PROPIEDAD RSP2

Si una descomposición $D = \{R_1, R_2, ..., R_m\}$ de R posee la propiedad de reunión sin pérdidas respecto a un conjunto de dependencias funcionales F sobre R, y si una descomposición $D_1 = \{Q_1, Q_2, ..., Q_k\}$ de R_i posee la propiedad de reunión sin pérdidas respecto a la *proyección* de F sobre R_i, entonces la descomposición $D_2 = \{R_1, R_2, ..., R_{i-1}, Q_1, Q_2, ..., Q_k, R_{i+1}, ... R_m\}$ de R posee la propiedad de reunión sin pérdidas respecto a F.

La propiedad RSP2 dice que, *si* una descomposición D ya posee la propiedad de reunión sin pérdidas —respecto a F— y descomponemos uno de los esquemas de relación R_i de D en otra descomposición D_1 que también tenga la propiedad de reunión sin pérdidas —respecto a $\pi_F(R_i)$—, *entonces* el reemplazo de R_i de D por D_1 ocasionará una descomposición que también habrá de poseer la propiedad de reunión sin pérdidas —respecto a F—. Supusimos implícitamente esta propiedad en los ejemplos de normalización informal de las secciones 12.3 y 12.4.

Con base en las propiedades RSP1 y RSP2 podemos elaborar el algoritmo 13.3 que crea una descomposición con reunión sin pérdidas D para R respecto a F tal que cada esquema de relación R_i en la descomposición D esté en FNBC.

ALGORITMO 13.3 Descomposición con reunión sin pérdidas para dar relaciones FNBC

1. hacer $D := \{ R \}$;
2. mientras haya un esquema de relación Q en D que no esté en FNBC hacer
   comenzar
   escoger un esquema de relación Q en D que no esté en FNBC;
   encontrar una dependencia funcional $X \rightarrow Y$ en Q que viole FNBC;
   reemplazar Q en D por dos esquemas $(Q - Y)$ y $(X \cup Y)$
   fin;

En cada repetición del ciclo del algoritmo 13.3 descomponemos un esquema de relación Q que no esté en FNBC para obtener dos esquemas de relación. En virtud de las propiedades RSP1 y RSP2, la descomposición D posee la propiedad de reunión sin pérdidas. Al final del algoritmo, todos los esquemas de relación en D estarán en FNBC. El lector puede comprobar que el ejemplo de normalización de las figuras 12.11 y 12.12 básicamente seguirá este

algoritmo. Las dependencias funcionales df3, df4 y, posteriormente, df5 violan FNBC, así que la relación LOTES se descompone correctamente en relaciones FNBC, y entonces la descomposición satisfará la propiedad de reunión sin pérdidas. En el paso 2 del algoritmo 13.3 es necesario determinar si un esquema de relación Q está en FNBC o no. Un método para lograrlo consiste en comprobar, para cada dependencia funcional $X \rightarrow Y$ en Q, si X^+ no incluye todos los atributos de Q. Si es así, $X \rightarrow Y$ violará FNBC, ya que en tal caso X no puede ser una (super)clave. Otra técnica se basa en la observación de que, siempre que un esquema de relación Q viole FNBC, existirá un par de atributos A y B en Q tales que $(Q - \{A, B\}) \rightarrow A$; si calculamos la cerradura $(Q - \{A, B\})^+$ para cada par de atributos $\{A, B\}$ de R, y si verificamos que la cerradura incluya a A (o a B), podremos determinar si Q está en FNBC.

Si queremos que una descomposición posea la propiedad de reunión sin pérdidas y conserve las dependencias, tendremos que conformarnos con esquemas de relación en 3FN, en vez de FNBC. Una simple modificación del algoritmo 13.1, que aparece en el algoritmo 13.4, produce una descomposición D de R que:

- Conserva las dependencias.
- Posee la propiedad de reunión sin pérdidas.
- Es tal que cada esquema de relación resultante en la descomposición está en 3FN.

ALGORITMO 13.4 Descomposición en esquemas de relación 3FN con reunión sin pérdidas y conservación de las dependencias

1. encontrar una cobertura mínima G para F;
   (* F es el conjunto de dependencias funcionales especificadas sobre R *)
2. para cada miembro izquierdo X que aparezca en G,
   crear un esquema de relación $\{X$ UNIÓN A_1 UNIÓN A_2 ... UNIÓN $A_m\}$,
   donde $X \rightarrow A_1$, $X \rightarrow A_2$, ..., $X \rightarrow A_m$ sean todas las dependencias en G
   con X como miembro izquierdo;
3. colocar cualesquier atributos restantes (no colocados) en un solo esquema de relación;
4. si ninguno de los esquemas de relación contiene una clave de R,
   crear un esquema de relación adicional que contenga atributos que formen una clave de R;

Puede demostrarse que la descomposición formada a partir del conjunto de esquemas de relación creado por el algoritmo anterior conservará las dependencias y poseerá la propiedad de reunión sin pérdidas. Es más, cada uno de los esquemas de relación en la descomposición está en 3FN. El paso 4 del algoritmo 13.4 implica identificar una clave de R. Podemos usar el algoritmo 13.4*a* para identificar una clave K de R basada en el conjunto de dependencias funcionales dadas. Comenzamos por igualar K al conjunto de todos los atributos de R; luego eliminamos un atributo a la vez y verificamos si los atributos restantes siguen formando una superclave. Observe que el conjunto de dependencias funcionales con el que se determina una clave en el algoritmo 13.4 podría ser F o bien G, puesto que son equivalentes. Observe también que el algoritmo 13.4*a* determina sólo *una clave para* R; la clave devuelta depende del orden en que se eliminen los atributos de R en el paso 2.

ALGORITMO 13.4*a* Obtención de una clave K para el esquema de relación R

1. hacer $K := R$;
2. para cada atributo A en K
   {calcular $(K - A)^+$ respecto al conjunto dado de dependencias funcionales;
   si $(K - A)^+$ contiene todos los atributos de R, hacer $K := K - \{A\}$ };

No siempre es posible encontrar una descomposición que conserve las dependencias y permita que cada esquema de relación de la descomposición esté en FNBC, más que en 3FN (como en el algoritmo 13.4). Podemos verificar individualmente los esquemas de relación de la descomposición para ver si satisfacen FNBC. Si algún esquema de relación R_i no está en FNBC, podemos optar por descomponerlo más o dejarlo en 3FN (con la posibilidad de algunas anomalías de actualización). El hecho de que no siempre podremos encontrar una descomposición en esquemas de relación en FNBC que conserve las dependencias puede ilustrarse con los ejemplos de la figura 12.12. Las relaciones LOTES1A (Fig. 12.12(a)) y R (Fig. 12.12(b)) no están en FNBC pero sí en 3FN. Cualquier intento por descomponer alguna de esas relaciones en relaciones FNBC ocasionará la pérdida de la dependencia df2 en LOTES1A o en R.

Es importante señalar que la teoría de las descomposiciones con reunión sin pérdidas se basa en la suposición de que *no se permiten valores* **nulos** *en los atributos de reunión*. En la siguiente sección analizaremos algunos de los problemas que pueden provocar los **nulos** en las descomposiciones relacionales.

13.1.4 *Problemas con valores nulos y tuplas colgantes*

Es preciso considerar con cuidado los problemas asociados a los **nulos** al diseñar un esquema de base de datos relacional. Todavía no existe una teoría de diseño relacional completamente satisfactoria que incluya valores **nulos**. Un problema se da cuando algunas tuplas tienen valores **nulos** en atributos que servirán para reunir relaciones individuales en la descomposición. Como ilustración, consideremos la base de datos de la figura 13.2(a), donde se muestran dos relaciones, EMPLEADO y DEPARTAMENTO. Las últimas dos tuplas de empleados —Bernal y Benítez— representan empleados recién contratados que todavía no se han asignado a un departamento (suponiendo que esto no viola ninguna restricción de integridad). Ahora suponga que deseamos obtener una lista de valores (NOMBREE, NOMBRED) para todos los empleados. Si aplicamos la operación * con EMPLEADO y DEPARTAMENTO (Fig. 13.2(b)), las dos tuplas antes mencionadas *no* aparecerán en el resultado. La operación de REUNIÓN EXTERNA, analizada en el capítulo 6, puede resolver este problema. Recordemos que, si obtenemos la REUNIÓN EXTERNA IZQUIERDA de EMPLEADO con DEPARTAMENTO, las tuplas de EMPLEADO que tienen **nulo** en el atributo de reunión sí aparecerán en el resultado, reunidas con una tupla "imaginaria" de DEPARTAMENTO que tiene **nulos** en todos sus valores de atributos. La figura 13.2(c) muestra el resultado.

En general, siempre que se diseñe un esquema de base de datos relacional en el que dos o más relaciones estén interrelacionadas a través de claves externas, debemos tener especial cuidado con los posibles valores nulos en dichas claves. Esto puede provocar una pérdida inesperada de información en las consultas que impliquen reuniones. Es más, si hay nulos en otros atributos, como SALARIO, su efecto sobre las funciones integradas como SUMA y PROMEDIO deberá evaluarse con mucho cuidado.

Un problema afín es el de las **tuplas colgantes**, que pueden presentarse si llevamos demasiado lejos una descomposición. Suponga que descomponemos la relación EMPLEADO de la figura 13.2(a) en EMPLEADO_1 y EMPLEADO_2, como en la figura 13.3(a) y 13.3(b).† Si aplicamos la operación * a EMPLEADO_1 y EMPLEADO_2, obtendremos la relación EMPLEADO original. Sin embargo, podríamos usar una representación alternativa para el caso en que un empleado todavía no se haya asignado a un departamento. Esto se ilustra en la figura 13.3(c), donde *no incluimos una tupla* en EMPLEADO_3 si el empleado todavía no se ha asignado a un departamento. Esta representación contrasta con la inclusión de una tupla con **nulo** en NÚMD, como en EMPLEADO_2. Suponga que usamos EMPLEADO_3 en vez de EMPLEADO_2. Ahora, si aplicamos una REUNIÓN NATURAL a EMPLEADO_1 y EMPLEADO_3, las tuplas de Bernal y Benítez desaparecerán. Éstas se llaman **tuplas colgantes** porque sólo se representan en una de las dos relaciones que representan empleados y, por tanto, se perderán si aplicamos una operación de reunión (interna).

13.1.5 *Análisis*

En las subsecciones precedentes presentamos varios algoritmo para diseñar bases de datos relacionales. Estos algoritmos descomponen un esquema de relación universal para producir un conjunto de esquemas de relación que se basan en los conceptos de dependencias funcionales, de formas normales, de conservación de dependencias y de descomposición con reunión sin pérdidas. También explicamos los problemas con los valores nulos y las tuplas colgantes en el diseño relacional.

Uno de los problemas con los algoritmos aquí descritos es que el diseñador de bases de datos debe especificar primero *todas* las dependencias funcionales pertinentes entre los atributos de la base de datos. Ésta *no es una tarea sencilla* en el caso de una base de datos grande con cientos de atributos. Si se omite la especificación de una o dos dependencias importantes el resultado puede ser un diseño deficiente. Otro problema es que estos algoritmos, en general, no sean deterministas. Por ejemplo, los algoritmos de síntesis requieren la especificación de una cobertura mínima para el conjunto de dependencias funcionales. Como en general puede haber muchas coberturas mínimas que correspondan a un conjunto de dependencias funcionales, el algoritmo puede producir diferentes diseños para el mismo conjunto de dependencias funcionales, dependiendo de la cobertura mínima que se use. Es posible que algunos de estos diseños no sean deseables. Otros algoritmos producen una descomposición que depende del orden en que se alimentan las dependencias funcionales al algoritmo; en este caso también pueden generarse muchos diseños diferentes que correspondan al mismo conjunto de dependencias funcionales.

Por las razones precedentes, estos algoritmos no pueden usarse a ciegas. Hasta ahora no han demostrado ser muy populares en la práctica; el diseño descendente de bases de datos apoyado en el modelo ER y otros modelos de datos de alto nivel se utiliza actualmente con mucha mayor frecuencia. Una técnica combina ambos enfoques. Por ejemplo, podemos comenzar con el modelo ER para producir un esquema conceptual y transformarlo a relaciones. Luego podemos aplicar los algoritmos a las relaciones individuales obtenidas del diseño ER, especificar sus dependencias funcionales y ver si aún necesitan descomponerse más. Otra alternativa es analizar los tipos de entidades del diseño ER y descomponerlos, si es necesario aplicando una teoría similar.

†Esto sucede a veces cuando aplicamos fragmentación vertical a una relación en el contexto de una base de datos distribuida (véase el Cap. 23).

(a) EMPLEADO

NOMBREE	NSS	FECHAN	DIRECCIÓN	NÚMD
Silva, José B.	123456789	09-ENE-55	Fresnos 731, Higueras, MX	5
Vizcarra, Federico T.	333445555	08-DIC-45	Valle 638, Higueras, MX	5
Zapata, Alicia J.	999887777	19-JUL-58	Castillo 3321, Sucre, MX	4
Valdés, Jazmín S.	987654321	20-JUN-31	Bravo 291, Belén, MX	4
Nieto, Ramón K.	666884444	15-SEP-52	Espiga 975, Heras, MX	5
Esparza, Josefa A.	453453453	31-JUL-62	Rosas 5631, Higueras, MX	5
Jabbar, Ahmed V.	987987987	29-MAR-54	Dalias 980, Higueras, MX	4
Botello, Jaime E.	888665555	10-NOV-27	Sorgo 450, Higueras, MX	1
Bernal, Andrés C.	999775555	26-ABR-55	Becerra 6530, Belén, MX	nulo
Benítez, Carlos M.	888664444	09-ENE-53	Bejuco 7654, Higueras, MX	nulo

DEPARTAMENTO

NOMBRED	NÚMD	NSSGTED
Investigación	5	333445555
Administración	4	987654321
Dirección	1	888665555

(b)

NOMBREE	NSS	FECHAN	DIRECCIÓN	NÚMD	NOMBRED	NSSGTED
Silva, José B.	123456789	09-ENE-55	Fresnos 731, Higueras, MX	5	Investigación	333445555
Vizcarra, Federico T.	333445555	08-DIC-45	Valle 638, Higueras, MX	5	Investigación	333445555
Zapata, Alicia J.	999887777	19-JUL-58	Castillo 3321, Sucre, MX	4	Administración	987654321
Valdés, Jazmín S.	987654321	20-JUN-31	Bravo 291, Belén, MX	4	Administración	987654321
Nieto, Ramón K.	666884444	15-SEP-52	Espiga 975, Heras, MX	5	Investigación	333445555
Esparza, Josefa A.	453453453	31-JUL-62	Rosas 5631, Higueras, MX	5	Investigación	333445555
Jabbar, Ahmed V.	987987987	29-MAR-59	Dalias 980, Higuera, MX	4	Administración	987654321
Botello, Jaime E.	888665555	10-NOV-27	Sorgo 450, Higueras, MX	1	Dirección	888665555

(c)

NOMBREE	NSS	FECHAN	DIRECCIÓN	NÚMD	NOMBRED	NSSGTED
Silva, José B.	123456789	09-ENE-55	Fresnos 731, Higueras, MX	5	Investigación	333445555
Vizcarra, Federico T.	333445555	08-DIC-45	Valle 638, Higueras, MX	5	Investigación	333445555
Zapata, Alicia J.	999887777	19-JUL-58	Castillo 3321, Sucre, MX	4	Administración	987654321
Valdés, Jazmín S.	987654321	20-JUN-31	Bravo 291, Belén, MX	4	Administración	987654321
Nieto, Ramón K.	666884444	15-SEP-52	Espiga 975, Heras, MX	5	Investigación	333445555
Esparza, Josefa A.	453453453	31-JUL-62	Rosas 5631, Higueras, MX	5	Investigación	333445555
Jabbar, Ahmed V.	987987987	29-MAR-59	Dalias 980, Higuera, MX	4	Administración	987654321
Botello, Jaime E.	888665555	10-NOV-27	Sorgo 450, Higueras, MX	1	Dirección	888665555
Bernal, Andrés C.	999775555	26-ABR-55	Becerra 6530, Belén, MX	nulo	nulo	nulo
Benitez, Carlos M.	888664444	09-ENE-53	Bejuco 7654, Higueras, MX	nulo	nulo	nulo

Figura 13.2 El problema de reunión con valores nulos. (a) Base de datos con nulos en algunos atributos de reunión. (b) Resultado de aplicar la operación de REUNIÓN NATURAL a las relaciones EMPLEADO y DEPARTAMENTO. (c) Resultado de aplicar la operación de REUNIÓN EXTERNA a EMPLEADO con DEPARTAMENTO.

(a) **EMPLEADO_1**

NOMBREE	NSS	FECHAN	DIRECCIÓN
Silva, José B.	123456789	09-ENE-55	Fresnos 731, Higueras, MX
Vizcarra, Federico T.	333445555	08-DIC-45	Valle 638, Higueras, MX
Zapata, Alicia J.	999887777	19-JUL-58	Castillo 3321, Sucre, MX
Valdés, Jazmín S.	987654321	20-JUN-31	Bravo 291, Belén, MX
Nieto, Ramón K.	666884444	15-SEP-52	Espiga 975, Heras, MX
Esparza, Josefa A.	453453453	31-JUL-62	Rosas 5631, Higueras, MX
Jabbar, Ahmed V.	987987987	29-MAR-54	Dalias 980, Higueras, MX
Botello, Jaime E.	888665555	10-NOV-27	Sorgo 450, Higueras, MX
Bernal, Andrés C.	999775555	26-ABR-55	Becerra 6530, Belén, MX
Benítez, Carlos M.	888664444	09-ENE-53	Bejuco 7654, Higueras, MX

(b) **EMPLEADO_2**

NSS	NÚMD
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1
999775555	nulo
888664444	nulo

(c) **EMPLEADO_3**

NSS	NÚMD
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987987987	4
888665555	1

Figura 13.3 El problema de la "tupla colgante". (a) La relación EMPLEADO_1, que incluye todos los atributos de la relación EMPLEADO excepto NÚMEROD. (b) La relación EMPLEADO_2, que incluye el atributo NÚMEROD de EMPLEADO con valores nulos. (c) La relación EMPLEADO_3, que no incluye las tuplas en las cuales NÚMEROD tiene un valor nulo.

Hasta ahora hemos analizado exclusivamente la dependencia funcional, que es, por mucho, el tipo de dependencias más importante en la teoría de diseño de bases de datos relacionales. No obstante, en muchos casos las relaciones tienen restricciones que no se pueden especificar como dependencias funcionales. En las siguientes secciones describiremos tipos adicionales de dependencias que servirán para representar otros tipos de restricciones sobre las relaciones. Algunas de estas dependencias conducen a otras formas normales. En la sección 13.2 veremos la dependencia multivaluada y definiremos la cuarta forma normal, que se basa en esta dependencia. Después, en la sección 13.3, trataremos brevemente las dependencias de reunión y la quinta forma normal. Por último, analizaremos someramente las dependencias de inclusión, las dependencias de patrón y la forma normal de dominio-clave.

13.2 Dependencias multivaluadas y cuarta forma normal

Las dependencias multivaluadas son una consecuencia de la primera forma normal, que prohíbe que un atributo de una tupla tenga un conjunto de valores. Si tenemos dos o más atributos multivaluados independientes en el mismo esquema de relación, nos enfrentaremos al

al problema de tener que repetir todos los valores de uno de los atributos con cada valor del otro atributo para que los ejemplares de la relación sigan siendo consistentes. Esta restricción se especifica con una dependencia multivaluada.

Por ejemplo, consideremos la relación EMP que se muestra en la figura 13.4(a). Una tupla de esta relación representa el hecho de que un empleado cuyo nombre es NOMBREE *trabaja en el proyecto* cuyo nombre es NOMBREPR y *tiene un dependiente* cuyo nombre es NOMBRED. Un empleado puede trabajar en varios proyectos y tener varios dependientes, y los proyectos y dependientes de un empleado no están relacionados directamente entre sí.† Para que las tuplas de la relación sean consistentes, deberemos tener una tupla por cada una de las

(a) **EMP**

NOMBREE	NOMBREPR	NOMBRED
Silva	X	Juan
Silva	Y	Ana
Silva	X	Ana
Silva	Y	Juan

(b) **PROYECTOS_EMP**

NOMBREE	NOMBREPR
Silva	X
Silva	Y

DEPENDIENTES_EMP

NOMBREE	NOMBRED
Silva	Juan
Silva	Ana

(c) **SUMINISTRAR**

NOMPROV	NOMRECOMP	NOMPROY
Silva	Perno	ProyX
Silva	Tuerca	ProyY
Aldama	Perno	ProyY
Velasco	Tuerca	ProyZ
Aldama	Clavo	ProyX
Aldama	Perno	ProyX
Silva	Perno	ProyY

(d) **R1**

NOMPROV	NOMBRECOMP
Silva	Perno
Silva	Tuerca
Aldama	Perno
Velasco	Tuerca
Aldama	Clavo

R2

NOMPROV	NOMPROY
Silva	ProyX
Silva	ProyY
Aldama	ProyY
Velasco	ProyZ
Aldama	ProyX

R3

NOMBRECOMP	NOMPROY
Perno	ProyX
Tuerca	ProyY
Perno	ProyY
Tuerca	ProyZ
Clavo	ProyX

Figura 13.4 4FN y 5FN. (a) La relación EMP con dos DMV: NOMBREE →» NOMBREPR y NOMBREE →» NOMBRED. (b) Descomposición de EMP en dos relaciones que están en 4FN. (c) La relación SUMINISTRAR, sin ninguna DMV, que está en 4FN (sin embargo, no estará en 5FN si se cumple la DR(R1,R2,R3). (d) Descomposición de la relación SUMINISTRAR con la dependencia de reunión para dar tres relaciones 5FN.

†En un diagrama ER, cada uno se representaría como un atributo multivaluado o como un tipo de entidades débil (véase el Cap. 3).

posibles combinaciones de dependiente y proyecto de un empleado. Esta restricción se especifica como una dependencia multivaluada sobre la relación EMP. En términos informales, siempre que en la misma relación se mezclan dos vínculos 1:*N independientes* A:B y A:C, puede surgir una DMV.

13.2.1 *Definición formal de dependencia multivaluada*

En términos formales, una **dependencia multivaluada** (DMV) X →» Y especificada sobre el esquema de relación R, donde X y Y son subconjuntos de R, especifica la siguiente restricción sobre cualquier relación *r* de R:

Si existen dos tuplas t_1 y t_2 en *r* tales que $t_1[X] = t_2[X]$, entonces deberán existir también dos tuplas t_3 y t_4 en *r* con las siguientes propiedades:†

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ y $t_4[Y] = t_2[Y]$.
- $t_3[R - (XY)] = t_2[R - (XY)]$ y $t_4[R - (XY)] = t_1[R - (XY)]$.

Siempre que se cumple X →» Y, decimos que X **multidetermina** Y. Debido a la simetría de la definición, siempre que X →» Y se cumple en R, también se cumple X →» (R – (XY)). Recordemos que R – (XY) es lo mismo que R – (X ∪ Y) = Z. Por tanto, X →» Y implica X Z, por lo que en ocasiones se escribe como X →» Y/Z.

La definición formal especifica que, dado un cierto valor de X, el conjunto de valores de Y determinado por este valor de X está determinado completamente por X solo, y *no depende* de los valores de los atributos restantes Z del esquema de relación R. Así pues, siempre que existan dos tuplas con distintos valores de Y pero el mismo valor de X, estos valores de Y deberán repetirse con cada valor distinto de Z que ocurra con ese mismo valor de X. De manera informal, esto equivale a que Y sea un atributo multivaluado de las entidades representadas por las tuplas de R.

En la figura 13.4(a) las DMV NOMBREE →» NOMBREPR y NOMBREE →» NOMBRED, o NOMBREE →» NOMBREPR/NOMBRED son válidas en la relación EMP. El empleado cuyo NOMBREE es 'Silva' trabaja en los proyectos cuyos NOMBREPR son 'X' y 'Y' y tiene dos dependientes cuyos NOMBRED son 'Juan' y 'Ana'. Si almacenamos sólo las dos primeras tuplas de EMP (<'Silva', 'X', 'Juan'> y <'Silva', 'Y', 'Ana'>), mostraremos, incorrectamente, asociaciones entre el proyecto 'X' y 'Juan' y entre el proyecto 'Y' y 'Ana'; esto no debe expresarse, porque no es el significado que se quiere de la relación. Así pues, deberemos almacenar las otras dos tuplas (<'Silva', 'X', 'Ana'> y <'Silva', 'Y', 'Juan'>) para hacer evidente que {'X', 'Y'} y {'Juan', 'Ana'} sólo están asociados a 'Silva'; esto es, que no hay asociación entre NOMBREPR y NOMBRED.

Una DMV X →» Y en R se denomina DMV **trivial** si (a) Y es un subconjunto de X o (b) X ∪ Y = R. Por ejemplo, la relación PROYECTOS_EMP de la figura 13.4(b) tiene la DMV trivial NOMBREE →» NOMBREPR. Una DMV que no satisface (a) ni (b) es una DMV **no trivial**. Una DMV trivial se cumple en *cualquier* ejemplar de relación *r* de R; se dice que es trivial porque no especifica ninguna restricción sobre R.

Si tenemos una DMV trivial en una relación, tal vez tengamos que repetir valores de manera redundante en las tuplas. En la relación EMP de la figura 13.4(a), los valores 'X' y 'Y' de NOMBREPR se repiten con cada valor de NOMBRED (o, simétricamente, los valores 'Juan'

†Las tuplas t_1, t_2, t_3 y t_4 no son necesariamente distintas.

y 'Ana' de NOMBRED se repiten con cada valor de NOMBREPR). Esta redundancia es a todas luces indeseable. Sin embargo, el esquema EMP está en FNBC porque no hay *ninguna* dependencia funcional que se cumpla en EMP. Por tanto, necesitamos definir una cuarta forma normal que sea más estricta que FNBC y prohíba los esquemas de relación del tipo de EMP. Primero estudiaremos algunas de las propiedades de las **DMV** y la manera en que se relacionan con las dependencias funcionales.

13.2.2 *Reglas de inferencia para las dependencias funcionales y multivaluadas*

Al igual que con las dependencias funcionales (DF), podemos desarrollar reglas de inferencia para las DMV. No obstante, es mejor desarrollar una armazón unificada que incluya tanto las DF como las DMV para poder considerar en conjunción ambos tipos de restricciones. Las siguientes reglas de inferencia, RI1 a RI8, constituyen un conjunto correcto y completo para inferir dependencias funcionales y multivaluadas a partir de un conjunto dado de dependencias. Supongamos que todos los atributos están incluidos en un esquema de relación "universal" $R = \{A_1, A_2, ..., A_n\}$ y que X, Y, Z y W son subconjuntos de R.

(RI1) (Regla reflexiva para DF): Si $X \subset Y$, entonces $X \rightarrow Y$.

(RI2) (Regla de aumento para DF): $\{X \rightarrow Y\} \varnothing XZ \rightarrow YZ$.

(RI3) (Regla transitiva para DF): $\{X \rightarrow Y, Y \rightarrow Z\} \varnothing X \rightarrow Z$.

(RI4) (Regla de complemento para DMV): $\{X \twoheadrightarrow Y\} \varnothing \{X \twoheadrightarrow (R - (X \cup Y))\}$.

(RI5) (Regla de aumento para DMV): Si $X \twoheadrightarrow Y$ y $W \subset Z$ entonces $WX \twoheadrightarrow YZ$.

(RI6) (Regla transitiva para DMV): $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \varnothing X \twoheadrightarrow (Z - Y)$.

(RI7) (Regla de réplica (DF a DMV)): $\{X \rightarrow Y\} \varnothing X \twoheadrightarrow Y$.

(RI8) (Regla de combinación para DF y DMV): Si $X \twoheadrightarrow Y$ y existe W con las propiedades de que (a) $W \cap Y$ está vacío, (b) $W \rightarrow Z$ y (c) $Y \subset Z$, entonces $X \rightarrow Z$.

RI1 a RI3 son las reglas de inferencia de Armstrong para DF exclusivamente. RI4 a RI6 son reglas de inferencia que atañen sólo a las DMV. RI7 y RI8 relacionan las DF con las DMV. En particular, RI7 establece que una dependencia funcional es un *caso especial* de una dependencia multivaluada; es decir, toda DF es también una DMV. Una DF $X \rightarrow Y$ es una DMV $X \twoheadrightarrow Y$ con la restricción adicional de que cuando más un valor de Y está asociado a cada valor de X. Dado un conjunto F de dependencias funcionales y multivaluadas especificado sobre $R = \{A_1, A_2, ..., A_n\}$, podemos usar RI1 a RI8 para inferir el conjunto (completo) de todas las dependencias (funcionales o multivaluadas) F^+ que se cumplirán en todos los ejemplares de relación r de R que satisfacen F. Aquí también llamamos a F^+ la **cerradura** de F.

13.2.3 *Cuarta forma normal (4FN)*

Ahora presentaremos la definición de 4FN, que se viola cuando una relación tiene dependencias multivaluadas indeseables y que, por tanto, puede usarse para identificar y descomponer tales relaciones. Un esquema de relación R está en **4FN** respecto a un conjunto de dependencias F si, para cada dependencia multivaluada *no trivial* $X \twoheadrightarrow Y$ en F^+, X es una superclave de R.

La relación EMP de la figura 13.4(a) no está en 4FN porque en las DMV no triviales NOMBREE $\twoheadrightarrow$ NOMBREPR y NOMBREE $\twoheadrightarrow$ NOMBRED, NOMBREE no es una superclave de

EMP. Descomponemos EMP en PROYECTOS_EMP y DEPENDIENTES_EMP, como se aprecia en la figura 13.4(b); estas dos nuevas relaciones están en 4FN porque NOMBREE $\twoheadrightarrow$ NOMBREPR es una DMV trivial en PROYECTOS_EMP y NOMBREE $\twoheadrightarrow$ NOMBRED es una DMV trivial en DEPENDIENTES_EMP. De hecho, no se cumple ninguna DMV no trivial en PROYECTOS_EMP ni en DEPENDIENTES_EMP. Tampoco se cumple ninguna DF en estos dos esquemas de relación.

A fin de ilustrar por qué es importante que las relaciones estén en 4FN, la figura 13.5(a) muestra la relación EMP con un empleado adicional, 'Bravo', que tiene tres dependientes ('Jaime', 'Juana' y 'Beto') y trabaja en cuatro proyectos distintos ('W', 'X', 'Y' y 'Z'). Hay 16 tuplas en la relación EMP de esta figura. Si descomponemos EMP en PROYECTOS_EMP y DEPENDIENTES_EMP, como en la figura 13.5(b), sólo necesitaremos almacenar un total de 11 tuplas en ambas relaciones. Por añadidura, estas tuplas serán mucho más pequeñas que las tuplas de EMP, y se evitarán las anomalías de actualización asociadas a las dependencias multivaluadas. Por ejemplo, si Bravo comienza a trabajar en otro proyecto, tendríamos que insertar tres tuplas más en EMP, una por cada dependiente. Si olvidáramos insertar alguna de ellas, la relación se volvería inconsistente al implicar, incorrectamente, un vínculo entre proyecto y dependiente. En cambio, sólo tendríamos que insertar una tupla en la relación 4FN PROYECTOS_EMP. Si una relación no está en 4FN, se originarán problemas similares con anomalías de eliminación y modificación.

La relación EMP de la figura 13.4(a) no está en 4FN, porque representa dos vínculos 1:*N* *independientes*: uno entre los empleados y los proyectos en los que trabajan, y otro entre los empleados y sus dependientes. Hay ocasiones en que tenemos un vínculo entre tres entidades que depende de las tres entidades participantes, como la relación SUMINISTRAR de la figura 13.4(c) (consideremos, por ahora, sólo las tuplas que están *arriba* de la línea punteada en esa figura). En este caso, una tupla representa un proveedor que suministra un componente específico *a un proyecto en particular*, de modo que *no* hay DMV no triviales. La relación SUMINISTRAR ya está *en 4FN* y no deberá descomponerse. Observe que todas las relaciones que contienen DMV no triviales tienden a ser relaciones de "sólo claves"; esto es, su clave consta de todos sus atributos juntos.

(a) **EMP**

NOMBREE	NOMBREPR	NOMBRED
Silva	X	Juan
Silva	Y	Ana
Silva	X	Ana
Silva	Y	Juan
Bravo	W	Jaime
Bravo	X	Jaime
Bravo	Y	Jaime
Bravo	Z	Jaime
Bravo	W	Juana
Bravo	X	Juana
Bravo	Y	Juana
Bravo	Z	Juana
Bravo	W	Beto
Bravo	X	Beto
Bravo	Y	Beto
Bravo	Z	Beto

(b) **PROYECTOS_EMP**

NOMBREE	NOMBREPR
Silva	X
Silva	Y
Silva	W
Bravo	X
Bravo	Y
Bravo	Z

DEPENDIENTES_EMP

NOMBREE	NOMBRED
Silva	Ana
Silva	Juan
Bravo	Jaime
Bravo	Juana
Bravo	Beto

Figura 13.5 Ventajas de la 4FN. (a) La relación EMP con algunas tuplas adicionales. (b) Proyección de EMP sobre PROYECTOS_EMP y DEPENDIENTES_EMP.

13.2.4 *Descomposición con reunión sin pérdidas para dar relaciones 4FN*

Siempre que descomponemos un esquema de relación R en $R_1 = (X \cup Y)$ y $R_2 = (R - Y)$ con base en una DMV $X \twoheadrightarrow Y$ que se cumple en R, la descomposición posee la propiedad de reunión sin pérdidas. Puede demostrarse que ésta es una condición necesaria y suficiente para descomponer un esquema en dos esquemas que posean la propiedad de reunión sin pérdidas, según la propiedad RSP1'.

PROPIEDAD RSP1'
Los esquemas de relación R_1 y R_2 forman una descomposición con reunión sin pérdidas de R si y sólo si $(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$ (o, por simetría, si y sólo si $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$).

Ésta es similar a la propiedad RSP1 de la sección 13.1.3, excepto que RSP1 sólo se refería a las DF, en tanto que RSP1' se refiere tanto a las DF como a las DMV (recuerde que una DF también es una DMV). Podemos modificar ligeramente el algoritmo 13.3 para producir el algoritmo 13.5, que crea una descomposición con reunión sin pérdidas para dar esquemas de relación que estén en 4FN (en vez de FNBC). El algoritmo 13.5 *no* produce necesariamente una descomposición que conserva las DF.

ALGORITMO 13.5 Descomposición con reunión sin pérdidas para dar relaciones 4FN

```
Hacer D := { R }
mientras haya un esquema de relación Q en D que no esté en 4FN hacer
    comenzar
    escoger un esquema de relación Q en D que no esté en 4FN;
    encontrar una DMV no trivial X ↠ Y en Q que viole 4FN;
    reemplazar Q en D por dos esquemas (Q – Y) y (X ∪ Y)
    fin;
```

13.3 Dependencias de reunión y quinta forma normal*

Vimos que RSP1 y RSP1' establecen la condición para descomponer un esquema de relación R en dos esquemas R_1 y R_2, donde la descomposición posee la propiedad de reunión sin pérdidas. Sin embargo, en algunos casos puede ser que no exista una descomposición con reunión sin pérdidas que dé dos esquemas de relación, pero sí que produzca más de dos esquemas de relación. Estos casos se manejan con la dependencia de reunión y la quinta forma normal. Es importante señalar que estos casos se presentan muy rara vez y que es difícil detectarlos en la práctica.

Una **dependencia de reunión** (DR), denotada por DR(R_1, R_2, ..., R_n), especificada sobre el esquema de relación R, especifica una restricción sobre los ejemplares r de R. La restricción establece que todo ejemplar permitido r de R debe tener una descomposición con reunión sin pérdidas para dar R_1, R_2, ..., R_n; esto es,

$$* (\pi_{<R1>}(r), \pi_{<R2>}(r), ..., \pi_{<Rn>}(r)) = r$$

Observe que una DMV es un caso especial de una DR donde $n = 2$. Una dependencia de reunión DR(R_1, R_2, ..., R_n), especificada sobre el esquema de relación R, es una DR **trivial** si

uno de los esquemas de relación R_i en DR(R_1, R_2, ..., R_n) es igual a R. Se dice que tal dependencia es trivial porque posee la propiedad de reunión sin pérdidas para *cualquier* ejemplar de relación r de R y, por tanto, no especifica ninguna restricción sobre R. Ahora podemos especificar la quinta forma normal, que también se denomina forma normal de proyección-reunión. Un esquema R está en **quinta forma normal** (5FN) [o **forma normal de proyección-reunión** (FNPR) (PJNF, *project-join normal form*)] respecto a un conjunto F de dependencias funcionales, multivaluadas y de reunión si, para cada dependencia de reunión no trivial DR(R_1, R_2, ..., R_n) en F^+ (esto es, implicada por F), toda R_i es una superclave de R.

Como ejemplo de DR, consideremos una vez más la relación SUMINISTRAR de la figura 13.4(c). Supongamos que siempre se cumple la siguiente restricción adicional: cuando un proveedor v suministra el componente c y *también* un proyecto p utiliza el componente c, y el proveedor v suministra *por lo menos un* componente al proyecto p, *entonces* el proveedor v suministra el componente c al proyecto p. Esta restricción puede expresarse de otras maneras, y especifica una dependencia de reunión DR(R1, R2, R3) entre las tres proyecciones R1(NOMPROV, NOMBRECOMP), R2(NOMPROV, NOMPROY) y R3(NOMBRECOMP, NOMPROY) de SUMINISTRAR. Si se cumple esta restricción, las tuplas que están abajo de la línea punteada en la figura 13.4(c) deberán existir en cualquier ejemplar permitido de la relación SUMINISTRAR que también contenga las tuplas que están arriba de dicha línea. La figura 13.4(d) muestra cómo se descompone la relación SUMINISTRAR *con la dependencia de reunión* para dar tres relaciones R1, R2 y R3 que están en 5FN. Adviértase que la aplicación de una REUNIÓN NATURAL a *cualesquiera dos* de estas relaciones produce tuplas espurias, pero la aplicación de REUNIÓN NATURAL a *las tres juntas* no lo hace. El lector deberá comprobar esto con el ejemplo de la relación de la figura 13.4(c) y sus proyecciones en la figura 13.4(d). Esto se debe a que sólo existe la DR, pero no se especifican DMV. Así mismo, cabe señalar que la DR(R1, R2, R3) se especifica sobre todos los ejemplares de relación permitidos, no sólo sobre el de la figura 13.4(c).

Es difícil descubrir dependencias de reunión en bases de datos reales con cientos de atributos; es por ello que en la práctica actual para el diseño de bases de datos se les presta poca atención.

13.4 Dependencias de inclusión*

Las dependencias de inclusión se definieron para formalizar las restricciones entre relaciones. Por ejemplo, la restricción de clave externa (o de integridad referencial) no puede especificarse como dependencia funcional o multivaluada porque relaciona atributos de varias relaciones; pero sí puede especificarse como una dependencia de inclusión.† En términos formales, una **dependencia de inclusión** $R.X < S.Y$ entre dos conjuntos de atributos —X del esquema de relación R y Y del esquema de relación S— especifica la restricción de que, si r es un estado de relación de R y s es un estado de relación de S, debe cumplirse:

$$\pi_{<X>}(r) \subseteq \pi_{<Y>}(s)$$

†Las dependencias de inclusión también pueden servir para representar la restricción entre dos relaciones que representen un vínculo clase/subclase de más alto nivel, lo que analizaremos en el capítulo 21.

El vínculo $\subseteq$ no tiene que ser un subconjunto propio. Por supuesto, los conjuntos de atributos sobre los que se especifica la dependencia de inclusión —X de R y Y de S— deben tener el mismo número de atributos. Además, los dominios de atributos correspondientes deben ser compatibles. Por ejemplo, si $X = \{A_1, A_2, ..., A_n\}$ y $Y = \{B_1, B_2, ..., B_n\}$, una posible correspondencia es que DOM(A_i) sea COMPATIBLE-CON DOM(B_i) para $1 \leq i \leq n$. En este caso decimos que A_i **se corresponde con** B_i.

Por ejemplo, podemos especificar las siguientes dependencias de inclusión sobre el esquema relacional de la figura 12.1:

DEPARTAMENTO.NSSGTED < EMPLEADO.NSS
TRABAJA_EN.NSS < EMPLEADO.NSS
EMPLEADO.NÚMEROD < DEPARTAMENTO.NÚMEROD
TRABAJA_EN.NÚMEROP < PROYECTO.NÚMEROP

Todas las dependencias de inclusión anteriores representan **restricciones de integridad referencial**. También podemos usar las dependencias de inclusión para representar **vínculos clase/subclase**. Por ejemplo, en el esquema relacional de la figura 21.12 (véase el Cap. 21), podemos especificar las siguientes dependencias de inclusión:

EMPLEADO.NSS < PERSONA.NSS
GRADOS_EXALUMNO.NSS < PERSONA.NSS
ESTUDIANTE.NSS < PERSONA.NSS

Existen reglas de inferencia para las dependencias de inclusión. Tres ejemplos de ellas son:

(RIDI1) $R.X < R.X$.
(RIDI2) Si $R.X < S.Y$, donde $X = \{A_1, A_2, ..., A_n\}$ y $Y = \{B_1, B_2, ..., B_n\}$ y A_i se corresponde con B_i, entonces $R.A_i < S.B_i$, para $1 \leq i \leq n$.
(RIDI3) Si $R.X < S.Y$ y $S.Y < T.Z$, entonces $R.X < T.Z$.

Se ha demostrado que las reglas de inferencia anteriores son correctas y completas para las dependencias de inclusión. Hasta ahora no se han desarrollado formas normales basadas en estas dependencias.

13.5 Otras dependencias y formas normales*

13.5.1 *Dependencias de patrón*

Por más tipos de dependencias que desarrollemos, siempre puede surgir alguna restricción peculiar que no se pueda representar con ninguna de ellas. La idea en que se basan las dependencias de patrón es la de especificar un patrón —o ejemplo— que defina cada restricción o dependencia. Hay dos tipos de patrones: generadores de tuplas y generadores de restricciones. Un patrón consiste en varias **tuplas de hipótesis** cuyo propósito es mostrar un ejemplo de las tuplas que pueden aparecer un una o más relaciones. La otra parte del patrón es la **conclusión del patrón**. En el caso de los patrones generadores de tuplas, la conclusión es un *conjunto de tuplas* que también deben existir en la relación si las tuplas de hipótesis están ahí. En el caso de los patrones generadores de restricciones, la conclusión del patrón es una *condición* que deben cumplir las tuplas de hipótesis.

La figura 13.6 muestra cómo podemos definir dependencias funcionales, multivaluadas y de inclusión con los patrones. La figura 13.7 ilustra cómo podemos especificar la restricción de que "el salario de un empleado no puede ser mayor que el de su supervisor directo" sobre el esquema de relación EMPLEADO de la figura 6.5.

13.5.2 *Forma normal de dominio-clave (FNDC)*

Siempre es posible definir formas normales más estrictas que tengan en cuenta tipos de dependencias y restricciones adicionales. La idea en que se basa la forma normal de dominio-clave es la de especificar (al menos en teoría) la "forma normal definitiva" que tiene en cuenta todos los posibles tipos de dependencias y restricciones. Se dice que una relación está en FNDC (*DKNF: domain-key normal form*) si todas las restricciones y dependencias que se debieran cumplir en la relación se pueden imponer con sólo hacer cumplir las restricciones de dominio y las restricciones de clave especificadas sobre la relación. Si una relación está en FNDC, resulta muy sencillo imponer las restricciones con sólo comprobar que todos los valores de atributos en una tupla pertenezcan al dominio apropiado y que se cumplan todas las restricciones de clave de la relación. Sin embargo, parece poco probable que se pueda incluir restricciones complejas en una relación FNDC, por lo que su utilidad práctica es limitada.

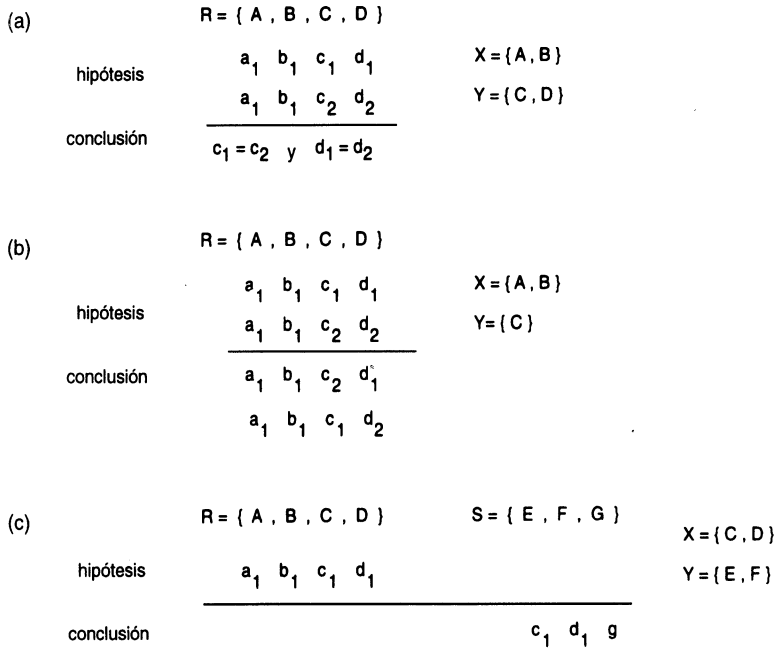


Figura 13.6 Patrones para tipos comunes de dependencias. (a) Patrón para la dependencia funcional $X \rightarrow Y$. (b) Patrón para la dependencia multivaluada $X \twoheadrightarrow Y$. (c) Patrón para la dependencia de inclusión $R.X < S.Y$.

EMPLEADO = { NOMBRE, NSS, ..., SALARIO, NSSSUPERVISOR }

	a	b	c	d
hipótesis	e	d	f	g
conclusión			c < f	

Figura 13.7 Patrón para la restricción de que el salario de un empleado debe ser menor que el de su supervisor.

13.6 Resumen

En la sección 13.1 examinamos el concepto de descomposición de una relación. Después presentamos algoritmos para descomponer un esquema de relación universal en un conjunto de esquemas de relación que están en formas normales y que satisfacen la propiedad de reunión sin pérdidas, la propiedad de conservación de las dependencias, o ambas propiedades, las cuales se basan en las dependencias funcionales especificadas sobre los atributos de la relación universal.

En seguida definimos otros tipos de dependencias y algunas formas normales más. Las dependencias multivaluadas nos llevaron a la cuarta forma normal, y las dependencias de reunión, a la quinta forma normal. También estudiamos las dependencias de inclusión, que sirven para especificar restricciones de integridad referencial y de clase/subclase, y las dependencias de patrón, con las que podemos especificar tipos de restricciones arbitrarios. Por último, tratamos de manera muy somera la forma normal de dominio-clave.

Preguntas de repaso

13.1. En una descomposición, ¿qué significa la condición de conservación de atributos?

13.2. ¿Por qué no bastan las formas normales por sí solas para asegurar un buen diseño de esquema?

13.3. ¿Qué es la propiedad de conservación de las dependencias de una descomposición? ¿Por qué es importante?

13.4. ¿Por qué no podemos garantizar que los esquemas de relación en una descomposición conservadora de las dependencias estén en FNBC?

13.5. ¿Qué es la propiedad de reunión sin pérdidas de una descomposición? ¿Por qué es importante?

13.6. Analice los problemas de los valores nulos y las tuplas colgantes.

13.7. ¿Qué es una dependencia multivaluada? ¿Qué tipo de restricción especifica? ¿Cuándo se presenta?

13.8. Defina la cuarta forma normal. ¿Por qué es útil?

13.9. Defina las dependencias de reunión y la quinta forma normal.

13.10. ¿Qué tipos de restricciones intentan representar las dependencias de inclusión?

13.11. ¿Qué diferencia hay entre las dependencias de patrón y los demás tipos de dependencias que vimos?

Ejercicios

13.12. Demuestre que los esquemas de relación producidos por el algoritmo 13.1 están en 3FN.

13.13. Demuestre que, si la matriz S producida por el algoritmo 13.2 no tiene una fila únicamente con símbolos "*a*", al proyectar S sobre la descomposición y volverla a reunir se originará siempre por lo menos una tupla espuria.

13.14. Demuestre que los esquemas de relación producidos por el algoritmo 13.3 están en FNBC.

13.15. Demuestre que los esquemas de relación producidos por el algoritmo 13.4 están en 3FN.

13.16. Especifique una dependencia de patrón para las dependencias de reunión.

13.17. Especifique todas las dependencias de inclusión para el esquema relacional de la figura 6.5.

13.18. Demuestre que una dependencia funcional también es una dependencia multivaluada.

13.19. Considere el ejemplo de normalizar la relación LOTES de la sección 12.4. Determine si la descomposición de LOTES en {LOTES1AX, LOTES1AY, LOTES1B, LOTES2} posee la propiedad de reunión sin pérdidas, aplicando el algoritmo 13.2.

13.20. Indique cómo pueden surgir las DMV NOMBREE $\twoheadrightarrow$ NOMBREPR y NOMBREE $\twoheadrightarrow$ NOMBRED de la figura 13.4(a) durante la normalización a 1FN de una relación, si los atributos NOMBREPR y NOMBRED son multivaluados (no simples).

13.21. Aplique el algoritmo 13.4*a* a la relación del ejercicio 12.28 para determinar una clave de R. Cree un conjunto mínimo de dependencias G que sea equivalente a F, y aplique el algoritmo de síntesis (algoritmo 13.4) para *descomponer* R en relaciones 3FN.

13.22. Repita el ejercicio 13.21 con las dependencias funcionales del ejercicio 12.29.

13.23. Aplique el algoritmo de descomposición (algoritmo 13.3) a la relación R y al conjunto de dependencias F del ejercicio 12.28. Repítalo con las dependencias G del ejercicio 12.29.

13.24. Aplique el algoritmo 13.4*a* a la relación del ejercicio 12.30 para determinar una clave de R. Aplique el algoritmo de síntesis (algoritmo 13.4) para descomponer R en relaciones 3FN, y el algoritmo de descomposición (algoritmo 13.3) para descomponer R en relaciones FNBC.

13.25. Escriba programas que implementen los algoritmos 13.3 y 13.4.

13.26. Considere las siguientes descomposiciones para el esquema de relación R del ejercicio 12.28. Determine si cada descomposición posee (i) la propiedad de conservación de las dependencias, y (ii) la propiedad de reunión sin pérdidas, respecto a F. Determine también en cuál forma normal está cada relación de la descomposición.

a. $D_1 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C\}$, $R_2 = \{A, D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$.

b. $D_2 = \{R_1, R_2, R_3\}$; $R_1 = \{A, B, C, D, E\}$, $R_2 = \{B, F, G, H\}$, $R_3 = \{D, I, J\}$.

c. $D_3 = \{R_1, R_2, R_3, R_4, R_5\}$; $R_1 = \{A, B, C, D\}$, $R_2 = \{D, E\}$, $R_3 = \{B, F\}$, $R_4 = \{F, G, H\}$, $R_5 = \{D, I, J\}$.

Bibliografía selecta

La teoría de la conservación de las dependencias y de las reuniones sin pérdidas se explica en el texto de Ullman (1988), donde aparecen demostraciones de algunos de los algoritmos que vimos aquí. La propiedad de reunión sin pérdidas se analiza en Aho *et al.* (1979). Los libros de Maier (1983) y Atzeni (1992) hacen un tratamiento muy completo de la teoría de la dependencia relacional.

El algoritmo de descomposición se debe a Bernstein (1976), y otros algoritmos de normalización se presentan en Biskup *et al.* (1979) y Tsou y Fischer (1982). En Osborn (1976) aparecen algoritmos para determinar las claves de una relación a partir de las dependencias funcionales; la comprobación de FNBC se analiza en Osborn (1979). La comprobación de 3FN se analiza en Jou y Fischer (1983). En Wang (1990) y en Hernández y Chan (1991) aparecen algoritmos para diseñar relaciones FNBC.

Las dependencias multivaluadas y la cuarta forma normal se definen en Zaniolo (1976) y en Fagin (1977). El conjunto de reglas correctas y completas para las dependencias funcionales y multivaluadas se expuso originalmente en Beeri *et al.* (1977). Las dependencias de reunión se analizan en Rissanen (1977) y Aho *et al.* (1979). Las reglas de inferencia para las dependencias de reunión se dan en Sciore (1982). La quinta forma normal (llamada forma normal de proyección-reunión) se presentó en Fagin (1979). Las dependencias de inclusión se explican en Casanova *et al.* (1981), y su empleo para optimizar esquemas relacionales se estudia en Casanova *et al.* (1989). En Sadri y Ullman (1982) se cubren las dependencias de patrón. Otras dependencias se estudian en Nicolas (1978), Furtado (1978), y Mendelzon y Maier (1979). La forma normal de dominio-clave se definió en Fagin (1981).

CAPÍTULO 14

Panorama del proceso de diseño de bases de datos

En este capítulo estudiaremos el proceso de diseño de bases de datos en sus diferentes fases. En el caso de bases de datos pequeñas que van a ser utilizadas por un número reducido de usuarios, el diseño no necesita ser muy complicado. Sin embargo, cuando se diseñan bases de datos medianas o grandes para el sistema de información de una organización muy grande, este proceso se vuelve bastante complejo, ya que el sistema debe satisfacer los requerimientos de muchos usuarios distintos. En tal caso, es imperativo que las fases de diseño y prueba aseguren que todos esos requerimientos se cumplan de manera satisfactoria. Es común que las bases de datos medianas y grandes las utilicen desde unos 25 hasta varios cientos de usuarios, que contengan millones de bytes de información e impliquen cientos de consultas y programas de aplicación. Tales bases de datos se usan ampliamente en organizaciones del gobierno, la industria y el comercio. Las industrias de servicios, como las bancarias, de energía eléctrica, agua, etc., de seguros, de transporte, hoteleras y de comunicaciones dependen totalmente de que sus bases de datos funcionen a la perfección las 24 horas del día. Los sistemas para estos tipos de aplicaciones suelen recibir el nombre de **sistemas de procesamiento de transacciones** por el gran número de transacciones que se aplican cotidianamente a la base de datos. Nos concentraremos aquí en el diseño de bases de datos para este tipo de aplicaciones. En el resto del capítulo, con *diseño de bases de datos* nos referiremos al proceso de diseño en un entorno de esta naturaleza. También veremos el lugar que ocupa una base de datos dentro del sistema de información de una organización grande y estudiaremos el ciclo de vida de un sistema de información representativo y de su sistema de base de datos componente.

En la sección 14.1 analizaremos por qué las bases de datos se han convertido en una parte importante de la gestión de recursos de información en muchas organizaciones, y examinaremos aspectos de su ciclo de vida. En la sección 14.2 trataremos las fases normales del diseño de bases de datos medianas o grandes. La sección 14.3 ofrece un panorama sobre el diseño de bases de datos físicas. Para concluir, en la sección 14.4 haremos un breve

examen de las herramientas de diseño automatizado. Las últimas dos secciones pueden pasarse por alto si no se desea un panorama detallado.

14.1 Papel de los sistemas de información en las organizaciones

14.1.1 *Contexto del empleo de sistemas de bases de datos en una organización*

A continuación estudiaremos, sin entrar en detalles, cómo es que los sistemas de bases de datos se han convertido en parte de los sistemas de información en muchas organizaciones. En la década de 1960, entre los sistemas de información predominaban los sistemas de archivos. Gradualmente, desde principios de los años setenta, las organizaciones han abandonado estos sistemas por las de bases de datos. Muchas organizaciones han creado departamentos dirigidos por un administrador de bases de datos (DBA: *database administrator*) para que supervise y controle las actividades que implica el ciclo de vida de las bases de datos. De manera similar, muchas organizaciones grandes reconocen que la gestión de recursos de información (IRM: *information resource management*) es decisiva para dirigir con éxito la empresa. Hay varias razones para ello:

- Más funciones de las organizaciones están computarizadas, lo que aumenta la necesidad de contar con grandes volúmenes de datos totalmente actualizados.
- Al crecer la complejidad de los datos y de las aplicaciones, se hace necesario modelar y mantener vínculos complejos entre los datos.
- Existe una tendencia hacia la consolidación de los recursos de información en muchas organizaciones.

En gran medida, los sistemas de bases de datos satisfacen los tres requerimientos anteriores. Dos características adicionales de estos sistemas son también muy valiosas para el diseño y manejo de bases de datos grandes:

- La *independencia con respecto a los datos* protege a los programas de aplicación contra cambios en la organización lógica primordial y en los caminos de acceso y estructuras de almacenamiento físicos.
- Los *esquemas externos* (vistas) permiten usar los mismos datos para múltiples aplicaciones, pues cada aplicación tiene su propia vista de los datos.

Una justificación adicional para cambiar a los sistemas de bases de datos es el bajo costo de crear nuevas aplicaciones en comparación con el costo de los sistemas de archivos precedentes. A menudo, esto justifica el alto costo inicial del diseño de la base de datos y de la conversión del sistema. La disponibilidad de lenguajes de acceso a datos de alto nivel simplifica la tarea de escribir aplicaciones de bases de datos, y los lenguajes de consulta de alto nivel hacen factible la consulta *ad hoc* de la información por parte de los gerentes de alto nivel.

Desde principios de los años setenta hasta mediados de los años ochenta, la tendencia fue hacia la creación de grandes depósitos centralizados de datos controlados por un solo SGBD centralizado. En fechas más recientes, esta tendencia se ha *invertido*, debido a los siguientes adelantos:

1. Los computadores personales y los productos de software similares a las bases de datos, como VISICALC, LOTUS 1-2-3, SYMPHONY, PARADOX y DBASE IV y V son muy utilizados por usuarios que antes caían en la categoría de usuarios esporádicos y ocasionales. Muchos administradores, ingenieros, científicos, arquitectos y otros similares pertenecen a esta categoría. En consecuencia, la práctica de crear **bases de datos personales** viene adquiriendo gran popularidad. Ahora es posible extraer una copia de parte de una base de datos grande de un computador central o de un servidor de bases de datos, trabajar con ella desde una estación de trabajo personal y volverla a almacenar en el computador central. De manera similar, los usuarios pueden diseñar y crear sus propias bases de datos y luego combinarlas con una más.
2. Con la aparición de los sistemas de gestión de bases de datos distribuidas (SGBDD; veáse el capítulo 23) ahora es posible la opción de distribuir las bases de datos entre múltiples sistemas de computador para mejorar el control local y acelerar el procesamiento local. Al mismo tiempo, los usuarios locales pueden tener acceso a datos remotos mediante los recursos con que cuenta el SGBDD.
3. Muchas organizaciones utilizan ahora **sistemas de diccionario de datos**, que son mini-SGBD que manejan los **metadatos** de un sistema de base de datos; esto es, los datos que describen la estructura, restricciones, aplicaciones, autorizaciones, etc., de la base de datos. A menudo se usan como *herramienta integral* para la gestión de recursos de información. Un sistema de diccionario de datos útil debe almacenar y controlar la siguiente información:
   a. Descripciones de los esquemas del sistema de base de datos.
   b. Información detallada sobre el diseño físico de la base de datos, como estructuras de almacenamiento, caminos de acceso y tamaños de archivos y registros.
   c. Descripciones de los usuarios de la base de datos, sus responsabilidades y sus derechos de acceso.
   d. Descripciones de alto nivel de las transacciones y aplicaciones de la base de datos, y de las interrelaciones entre los usuarios y las transacciones.
   e. La relación entre las transacciones de la base de datos y los elementos de información a los que hacen referencia. Esto resulta útil para determinar cuáles transacciones serán afectadas cuando se modifiquen ciertas definiciones de los datos.
   f. Cifras estadísticas de utilización, como las frecuencias de consultas y transacciones y el número de accesos a diferentes porciones de la base de datos.

   Esta información está disponible para administradores de bases de datos, diseñadores y usuarios autorizados en forma de documentación en línea. Esto mejora el control de los administradores de bases de datos sobre el sistema de información, y la comprensión y el aprovechamiento del sistema por parte de los usuarios. En muchas organizaciones grandes, el sistema de diccionario de datos se considera tan importante como un SGBD.

En fechas recientes, se ha hecho hincapié en los **sistemas de procesamiento de transacciones** de alto rendimiento, que requieren una operación continua las 24 horas y se utilizan en la industria de servicios. Es común que cientos de transacciones, provenientes de

terminales remotas y locales, tengan acceso a estas bases de datos cada minuto. El rendimiento de las transacciones, en términos del promedio de transacciones por minuto y del tiempo de respuesta medio y máximo a las transacciones, es decisvo en estas aplicaciones. En estos tipos de sistemas es indispensable un diseño cuidadoso de bases de datos físicas que satisfaga las necesidades del procesamiento de transacciones para la organización.

Algunas organizaciones han subordinado su gestión de recursos de información a ciertos productos de SGBD y de diccionario de datos. Su inversión en el diseño e implementación de sistemas muy grandes y complejos les dificulta mucho cambiar a productos de SGBD más nuevos, y así la organización "se ha comprometido" con su sistema de SGBD actual. En lo tocante a tales bases de datos siempre valdrá la pena insistir en la importancia de cuidar el diseño para que tenga en cuenta la necesidad de una posible modificación del sistema en el futuro si hay cambios en los requerimientos. El costo puede ser muy alto si un sistema grande y complejo no puede evolucionar y se hace necesario sustituirlo por otros productos de SGBD.

14.1.2 *Ciclo de vida de un sistema de información*

En una organización grande, el sistema de base de datos suele ser parte de un sistema de información mucho mayor con el que aquella controla sus recursos de información. Un **sistema de información** incluye todos los recursos dentro de la organización que participan en la recolección, administración, uso y diseminación de la información. En un entorno computarizado, estos recursos incluirán los datos mismos, el SGBD, el hardware del computador y los medios de almacenamiento, el personal que usa y maneja los datos (DBA, usuarios finales, usuarios paramétricos, etc.), el software de aplicación que tiene acceso a los datos y los actualiza, y los programadores que crean estas aplicaciones. Así pues, el sistema de base de datos es sólo una parte de un sistema de información mucho mayor dentro de la organización.

En esta sección examinaremos un ciclo de vida representativo de un sistema de información y veremos el lugar que ocupa el sistema de base de datos dentro de este ciclo de vida. A menudo se llama **macro ciclo de vida** al ciclo de vida del sistema de información, y **micro ciclo de vida** al ciclo de vida del sistema de base de datos. Por lo regular, el macro ciclo de vida incluye las siguientes fases:

1. Análisis de factibilidad: Éste se ocupa de analizar las posibles áreas de aplicación, realizar estudios preliminares de costo-beneficios y establecer prioridades entre las aplicaciones.
2. Recolección y análisis de requerimientos: Se obtienen requerimientos detallados interactuando con los posibles usuarios para identificar sus problemas y necesidades específicos.
3. Diseño: Esta fase tiene dos aspectos: El diseño del sistema de base de datos y el diseño de los sistemas de aplicación (programas) que usan y procesan la base de datos.
4. Implementación: El sistema de información se implementa, la base de datos se carga y las transacciones de la base de datos se implementan y prueban.
5. Validación y prueba de aceptación: Se valida la aceptabilidad del sistema en cuanto a la satisfacción de los requerimientos de los usuarios y a los criterios de rendimiento. El sistema se prueba contra los criterios de rendimiento y las especificaciones de comportamiento.
6. Operación: Ésta puede ir precedida por la conversión de los usuarios de un sistema anterior así como por la capacitación de los usuarios. La fase operativa comienza cuando todas las funciones del sistema están disponibles y han sido validadas. Al surgir nuevos requerimientos o aplicaciones, pasan por todas las fases anteriores hasta validarse e incorporarse al sistema. La supervisión del rendimiento del sistema y el mantenimiento del sistema son actividades importantes durante la fase de operación.

14.1.3 *Ciclo de vida del sistema de aplicación de base de datos*

Entre las actividades relacionadas con el (micro) ciclo de vida del sistema de aplicación para la base de datos están las siguientes fases:

1. Definición del sistema: Se definen el alcance del sistema de base de datos, sus usuarios y sus aplicaciones.
2. Diseño: Al final de esta fase, estará listo un diseño lógico y físico completo del sistema de base de datos en el SGBD elegido.
3. Implementación: Esto comprende el proceso de escribir las definiciones conceptual, externa e interna de la base de datos, crear archivos de base de datos vacíos e implementar las aplicaciones de software.
4. Carga o conversión de los datos: La base de datos se alimenta ya sea cargando los datos directamente o convirtiendo archivos ya existentes al formato del sistema de base de datos.
5. Conversión de aplicaciones: Cualesquier aplicaciones de software que se usaban en un sistema anterior se convierten al nuevo sistema.
6. Prueba y validación: Se prueba y valida el nuevo sistema.
7. Operación: El sistema de base de datos y sus aplicaciones se ponen en operación.
8. Supervisión y mantenimiento: Durante la fase de operación, el sistema se vigila y mantiene constantemente. Puede haber crecimiento y expansión tanto en el contenido de datos como en las aplicaciones de software. Es posible que de vez en cuando se requieran modificaciones y reorganizaciones importantes.

Las actividades 2, 3 y 4 juntas forman parte de las fases de diseño e implementación del ciclo de vida del sistema de información. En la sección 14.2 destacamos la actividad 2, que cubre la fase de diseño para la base de datos. Casi todas las bases de datos en las organizaciones experimentan todas las actividades del ciclo de vida anterior. Los pasos de conversión (4 y 5) no son aplicables cuando tanto la base de datos como las aplicaciones son nuevas. Cuando una organización pasa de un sistema antiguo bien establecido a uno nuevo, las actividades 4 y 5 tienden a ser las más prolongadas, y es común que se subestime el esfuerzo requerido para llevarlas a cabo. En general, suele haber retroalimentación entre los diversos pasos porque a menudo surgen nuevos requerimientos en todas las etapas.

14.2 El proceso de diseño de bases de datos

Ahora nos concentraremos en el paso 2 del ciclo de vida del sistema de aplicación de base de datos, al que llamaremos diseño de bases de datos. El problema del diseño de bases de datos puede expresarse así: *diseñar la estructura lógica y física de una o más bases de datos para atender las necesidades de información de los usuarios en una organización para un conjunto definido de aplicaciones.*

Las *metas* de un diseño de bases de datos son múltiples: satisfacer los requerimientos de *contenido* de información de los usuarios y aplicaciones especificados; proveer una *estructuración* de la información natural y fácil de entender, y apoyar los *requerimientos de procesamiento* y cualesquier otros objetivos de rendimiento, como el tiempo de respuesta, el tiempo de procesamiento y el espacio de almacenamiento. Es muy difícil lograr y medir estas metas. El problema se agrava porque el proceso de diseño de bases de datos a menudo comienza con requerimientos muy informales y muy mal definidos. En contraste, el resultado de la actividad de diseño es un esquema de base de datos rígidamente definido que no se podrá modificar fácilmente una vez implementada la base de datos. Podemos identificar seis fases principales del proceso de diseño de bases de datos:

1. Recolección y análisis de requerimientos.
2. Diseño conceptual de la base de datos.
3. Elección de un SGBD.
4. Transformación al modelo de datos (llamado también diseño lógico de la base de datos).
5. Diseño físico de la base de datos.
6. Implementación del sistema de base de datos.

El proceso de diseño consta de dos actividades paralelas, como se ilustra en la figura 14.1. La primera actividad implica el diseño del **contenido de datos y estructura** de la base de datos; la segunda atañe el diseño del **procesamiento de la base de datos y de las aplicaciones de software**. Estas dos actividades están íntimamente entrelazadas. Por ejemplo, podemos identificar los elementos de información que se almacenarán en la base de datos analizando las aplicaciones de esta última. Además, la fase de diseño físico de la base de datos, durante la cual elegimos las estructuras de almacenamiento y los caminos de acceso de los archivos de la base de datos, depende de las aplicaciones que van a utilizar estos archivos. Por otro lado, casi siempre especificamos el diseño de las aplicaciones de base de datos haciendo referencia a los elementos que tiene el esquema de la base de datos, que han de especificarse durante la primera actividad. Es evidente que entre estas dos actividades hay una fuerte influencia mutua. Tradicionalmente, las metodologías de diseño de bases de datos se han concentrado sobre todo en una de estas actividades o en la otra; ello podría calificarse como diseño de bases de datos **controlado por los datos** o **controlado por los procesos**. Hoy día se reconoce que ambas actividades deben efectuarse en coordinación.

Las seis fases que acabamos de mencionar no tienen que realizarse en una secuencia estricta. En muchos casos es posible que nos veamos obligados a modificar el diseño de una fase anterior durante una fase subsecuente. Estos **ciclos de retroalimentación** entre fases —y también dentro de las fases— son comunes durante el diseño de bases de datos. No

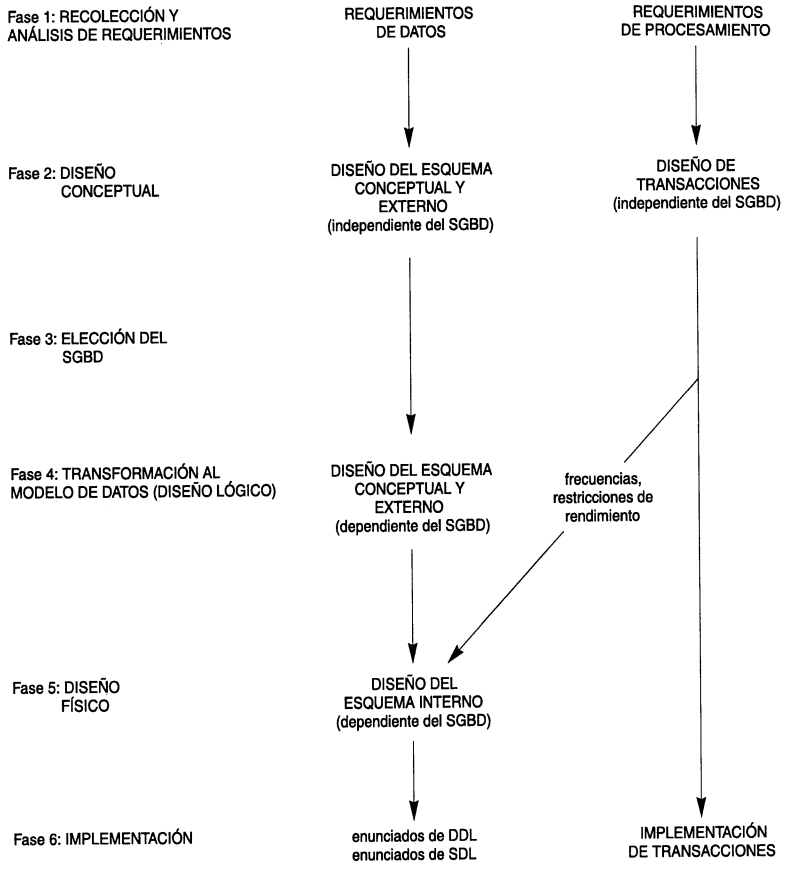


Figura 14.1 Fases del diseño de bases de datos grandes.

mostraremos los ciclos de retroalimentación en la figura 14.1 para que el diagrama no se complique. La fase 1 de dicha figura implica recabar información sobre el uso que se piensa dar a la base de datos, en tanto que la fase 6 se ocupa de la implementación de la base de datos. A veces se considera que las fases 1 y 6 *no* forman parte del diseño de bases de datos propiamente dicho, sino que son parte del ciclo de vida del sistema de información, más general. El corazón del proceso de diseño de bases de datos lo constituyen las fases 2, 4 y 5, que resumiremos brevemente aquí:

- *Diseño conceptual de la base de datos (fase 2):* La meta de esta fase es producir un esquema conceptual de la base de datos que sea independiente de un SGBD específico.

A menudo usamos un modelo de datos de alto nivel como el modelo ER o el EER (véase el Cap. 21) durante esta fase. Además, especificamos tantas de las aplicaciones o transacciones conocidas de la base de datos como sea posible, empleando una notación que es independiente de cualquier SGBD específico.

- *Transformación al modelo de datos (fase 4):* Esto se denomina también **diseño lógico de la base de datos**. Durante esta fase **transformamos** el esquema conceptual del modelo de datos de alto nivel empleado en la fase 2 al modelo de datos del SGBD elegido en la fase 3. Podemos iniciar esta fase después de escoger un modelo de datos de implementación, sin esperar la elección de un SGBD específico; por ejemplo, si decidimos usar algún SGBD relacional pero todavía no hemos escogido uno en particular. Llamamos a esto diseño lógico *independiente del sistema* (pero *dependiente del modelo de datos*). En términos de la arquitectura del SGBD de tres niveles que vimos en el capítulo 2, el resultado de esta fase es un *esquema conceptual* en el modelo de datos elegido. Además, es común efectuar durante esta fase el diseño de *esquemas externos* (vistas) para aplicaciones específicas.
- *Diseño físico de la base de datos (fase 5):* Durante esta fase diseñamos las especificaciones para la base de datos almacenada en términos de estructuras de almacenamiento físicas, colocación de registros y caminos de acceso. Esto corresponde a diseñar el *esquema interno* en la terminología de la arquitectura del SGBD de tres niveles.

En las subsecciones que siguen analizaremos cada una de las seis fases correspondientes al diseño de bases de datos.

14.2.1 *Fase 1: Recolección y análisis de requerimientos*

Antes de poder diseñar eficazmente una base de datos, debemos conocer las expectativas de los usuarios y los usos que se piensa dar a la base de datos con el mayor detalle posible. El proceso de identificar y analizar los usos propuestos se denomina **recolección y análisis de requerimientos**. Para especificar los requerimientos, primero debemos identificar las demás partes del sistema de información que van a interactuar con el sistema de base de datos. Entre ellas están los usuarios y las aplicaciones nuevos y ya existentes. Una vez hecho esto, se reunirán y analizarán los requerimientos de dichos usuarios y aplicaciones. Por lo regular, esta fase incluye las siguientes actividades:

- Identificación de las principales áreas de aplicación y grupos de usuarios que utilizarán la base de datos. Se eligen individuos clave dentro de cada grupo como participantes destacados en los pasos subsecuentes de recolección y especificación de requerimientos.
- Estudio y análisis de la documentación existente relativa a las aplicaciones. Se repasa otra documentación —manuales de políticas, formas, informes y diagramas de organización— para determinar si influye de alguna manera sobre el proceso de recolección y especificación de requerimientos.
- Estudio del entorno de operación actual y de los planes de aprovechamiento de la información. Esto incluye el análisis de los tipos de transacciones y de sus frecuencias, así como del flujo de información dentro del sistema. Se especifican los datos de entrada y salida de las transacciones.

- Recolección de respuestas escritas a grupos de preguntas hechas a los posibles usuarios de la base de datos. Estas preguntas se refieren a las prioridades de los usuarios y a la importancia que dan a las diversas aplicaciones. Posiblemente se entrevisten individuos clave que ayudarán a estimar el valor de la información y a establecer las prioridades.

Los métodos anteriores para recabar los requerimientos producen especificaciones de requerimientos mal estructuradas y en su mayor parte informales, las cuales se convierten después a una forma más estructurada mediante una de las **técnicas de especificación de requerimientos** más formales. Éstas incluyen los diagramas HIPO (*hierarchical input process output*: entrada/procesamiento/salida jerárquicos), SADT (*structured analysis and design technique*: técnica de análisis y diseño estructurados), DFD (*data flow diagrams*: diagramas de flujo de datos), diagramas Orr-Warnier y diagramas Nassi-Schneiderman. Todos estos son métodos diagramáticos para organizar y presentar los requerimientos de procesamiento de la información. Los diagramas pueden adoptar la forma de jerarquías, diagramas de flujo, estructuras secuenciales y cíclicas, etc., dependiendo del método. Estos diagramas suelen ir acompañados por documentación adicional en forma de texto, tablas, gráficas y requerimientos de decisión. Hay libros enteros que tratan la fase de recolección y análisis de requerimientos (véanse las notas bibliográficas al final del capítulo). Esta fase puede requerir bastante tiempo, pero es crucial para el éxito futuro del sistema de información.

Se han propuesto algunas técnicas asistidas por computador para ocuparse de la recolección y el análisis de requerimientos. Estas técnicas incluyen herramientas automatizadas para el análisis de requerimientos, para comprobar que las especificaciones son consistentes y completas. Los requerimientos se almacenan en un depósito único, usualmente llamado base de datos de diseño, y se pueden exhibir y actualizar conforme avanza el diseño. Uno de los primeros ejemplos es el lenguaje de especificación de problemas y el analizador de especificación de problemas (PSL/PSA: *Problem Statement Language/Problem Statement Analyzer*) creado por el proyecto ISDOS en la University of Michigan (véanse las notas bibliográficas).

14.2.2 *Fase 2: Diseño conceptual de la base de datos*

La segunda fase del diseño de base de datos implica dos actividades paralelas. La primera, el **diseño del esquema conceptual**, examina los requerimientos de datos resultantes de la fase 1 y produce un esquema de base de datos conceptual. La segunda actividad, el **diseño de transacciones**, examina las aplicaciones de base de datos analizadas en la fase 1 y produce especificaciones de alto nivel para estas transacciones.

Fase 2a: Diseño del esquema conceptual. El esquema conceptual que resulta de esta fase suele estar contenido en un modelo de datos de alto nivel independiente del SGBD, de modo que no puede usarse directamente para implementar la base de datos. La importancia de un esquema conceptual independiente del SGBD no puede sobreestimarse, por las siguientes razones:

1. La meta del diseño del esquema conceptual es un entendimiento completo de la estructura, el significado (semántica), los vínculos y las restricciones de la base de datos. Lo mejor es lograr esto independientemente de un SGBD específico porque todos los SGBD suelen tener peculiaridades y restricciones que no deben influir sobre el diseño del esquema conceptual.

2. El esquema conceptual es muy valioso como una *descripción estable* del contenido de la base de datos. La elección del SGBD y las decisiones de diseño posteriores pueden cambiar, sin alterar el esquema conceptual independiente del SGBD.
3. Un buen entendimiento del esquema conceptual es decisivo para los usuarios de la base de datos y los diseñadores de aplicaciones. Por ello es sumamente importante el empleo de un modelo de datos de alto nivel que sea más expresivo y general que los modelos de datos de los SGBD individuales.
4. La descripción diagramática del esquema conceptual puede servir como un excelente vehículo de comunicación entre los usuarios, diseñadores y analistas de la base de datos. Como los modelos de datos de alto nivel suelen depender de conceptos que son más fáciles de entender que los modelos de datos de más bajo nivel, específicos para cada SGBD, cualquier comunicación referente al diseño del esquema se hace más exacta y directa.

En esta fase del diseño, es importante usar un modelo de datos de alto nivel —también denominado modelo de datos semántico o conceptual— que tenga las siguientes características:

1. Expresividad: El modelo de datos debe ser lo bastante expresivo para distinguir los diferentes tipos de datos, vínculos y restricciones.
2. Sencillez: El modelo debe ser lo bastante simple para que la generalidad de los usuarios no especialistas comprendan y usen sus conceptos.
3. Minimalidad: El modelo debe tener un número pequeño de conceptos básicos cuyo significado sea distinto y no se traslape.
4. Representación diagramática: El modelo debe contar con una representación diagramática para mostrar un esquema conceptual que sea fácil de interpretar.
5. Formalidad: Un esquema conceptual expresado en el modelo de datos debe representar una especificación formal, sin ambigüedad, de los datos. Por tanto, los conceptos del modelo deben definirse con exactitud y sin que haya posibilidad de confusión.

En ocasiones estos requerimientos entrarán en conflicto. En particular, el requerimiento 1 entra en conflicto con los demás. Se han propuesto muchos modelos conceptuales de alto nivel para el diseño de bases de datos (véase la bibliografía seleccionada para el capítulo 21). En el análisis que sigue, emplearemos la terminología del modelo de entidad-vínculo extendido (EER: *Enhanced Entity-Relationship*) que se presenta en los capítulos 3 y 21, y supondremos que se está utilizando en esta fase.

ENFOQUES PARA EL DISEÑO DE ESQUEMAS CONCEPTUALES: Para diseñar un esquema conceptual debemos identificar los componentes básicos del esquema: los tipos de entidades, los tipos de vínculos y sus atributos. También debemos especificar los atributos clave, la cardinalidad y las restricciones de participación en vínculos, tipos de entidades débiles, y jerarquías y retículas de especialización/generalización (si es necesario). Este diseño se deriva de los requerimientos recabados durante la fase 1.

Hay dos enfoques para diseñar el esquema conceptual. En el primero, al que llamaremos **enfoque centralizado** (o **de una vez por todas**) **de diseño del esquema**, los requerimientos de las diferentes aplicaciones y grupos de usuarios de la fase 1 se combinarán en un solo

de las diferentes aplicaciones y grupos de usuarios de la fase 1 se combinarán en un solo conjunto de requerimientos *antes de iniciarse el diseño del esquema*. A continuación se diseña un solo esquema que corresponde al conjunto combinado de requerimientos. Cuando hay muchos usuarios, la combinación de todos los requerimientos puede ser una tarea ardua y muy tardada. La suposición en que se basa este enfoque es que una autoridad centralizada, el DBA, se encarga de decidir cómo combinar los requerimientos de los diferentes usuarios y aplicaciones, y de diseñar el esquema conceptual para toda la base de datos. Una vez diseñado y terminado el esquema conceptual, el DBA puede especificar esquemas externos para los diferentes grupos de usuarios y aplicaciones.

En el segundo enfoque, al que llamamos **enfoque de integración de vistas**, no combinamos los requerimientos; más bien, diseñamos un esquema (o vista) para cada grupo de usuarios o aplicación con base sólo en sus requerimientos. Como resultado, creamos un esquema (vista) de alto nivel para cada uno de esos grupos de usuarios o aplicaciones. Durante una fase subsecuente de **integración de vistas**, estos esquemas se combinan o integran para formar un **esquema conceptual global** para toda la base de datos. Las vistas individuales pueden reconstruirse como esquemas externos después de la integración de las vistas.

La diferencia principal entre los dos enfoques radica en la manera y las circunstancias en que las diferentes vistas o requerimientos de los múltiples usuarios y aplicaciones se concilian y combinan. En el enfoque centralizado, la reconciliación la efectúa manualmente el personal del DBA antes de diseñar algún esquema, y se aplica directamente a los requerimientos recabados en la fase 1. Este enfoque es el que se ha utilizado tradicionalmente a pesar de la carga que representa para el personal del DBA conciliar las diferencias y conflictos entre los grupos de usuarios para establecer una definición clara de los requerimientos globales *antes* de intentar el diseño del esquema conceptual. Debido a la dificultad inherente a esta tarea, el enfoque de integración de vistas goza cada vez de mayor aceptación.

En el enfoque de integración de vistas, cada grupo de usuarios o aplicación diseña realmente su propio esquema conceptual (EER) a partir de sus requerimientos. Luego el DBA aplica un proceso de integración a estos esquemas (vistas) para formar el esquema global integrado. Aunque la integración de vistas puede efectuarse manualmente, su aplicación a una base de datos grande en la que intervengan decenas de grupos de usuarios requerirá una metodología y el empleo de herramientas automatizadas que ayuden a realizar la integración. Las correspondencias entre atributos, tipos de entidades y tipos de vínculos en las diversas vistas deben especificarse antes de que pueda aplicarse la integración. Por añadidura, hay que resolver problemas como por ejemplo la integración de vistas conflictivas y la verificación de la consistencia de las correspondencias especificadas entre los esquemas.

ESTRATEGIAS PARA EL DISEÑO DE ESQUEMAS: Dado un conjunto de requerimientos, sea para un solo usuario o para una comunidad de usuarios grande, debemos crear un esquema conceptual que satisfaga dichos requerimientos. Hay varias estrategias para diseñar tales esquemas, y la mayoría de ellas seguirán un enfoque incremental; es decir, parten de ciertas construcciones de esquema derivadas de los requerimientos y luego modifican, refinan o desarrollan de manera incremental dichas construcciones. Ahora analizaremos algunas de estas estrategias:

1. Estrategia descendente: Se parte de un esquema que contiene abstracciones de alto nivel y luego se aplican refinaciones descendentes sucesivas. Por ejemplo, podemos comenzar por especificar sólo unos cuantos tipos de entidades de alto nivel; luego,

al especificar sus atributos, los dividimos en tipos de entidades y de vínculos de menor nivel. El proceso de especialización para refinar un tipo de entidades convirtiéndolo en subclases (véase la Sec. 21.1) es otro ejemplo de estrategia de diseño descendente.

2. Estrategia ascendente: Se parte de un esquema que contiene abstracciones básicas, y luego se combinan o se les añaden otras abstracciones. Por ejemplo, podríamos comenzar con los atributos y agruparlos en tipos de entidades y vínculos. Conforme avanza el diseño, podríamos añadir nuevos vínculos entre tipos de entidades. El proceso de generalizar subclases para obtener clases generalizadas de más alto nivel (véase la Sec. 21.1) es otro ejemplo de estrategia de diseño ascendente.
3. Estrategia de adentro hacia afuera: Éste es un caso especial de estrategia ascendente, en la que la atención se concentra en un conjunto central de conceptos que son los más evidentes. A continuación el modelado *se extiende hacia afuera* al considerar conceptos nuevos en las cercanías de los ya existentes. Por ejemplo, podríamos especificar en el esquema unos cuantos tipos de entidades obvios y continuar agregando otros tipos de entidades y de vínculos relacionados con ellos.
4. Estrategia mixta: En vez de seguir una estrategia específica durante todo el diseño, los requerimientos se dividirán según una estrategia descendente, y se diseñará una parte del esquema para cada partición de acuerdo con una estrategia ascendente. Por último, se combinarán las diferentes partes del esquema.

Las figuras 14.2 y 14.3 ilustran las refinaciones descendente y ascendente, respectivamente. Un ejemplo de primitiva de refinación descendente es la descomposición de un tipo de entidades en varios tipos. La figura 14.2(a) muestra cómo CURSO se refina en CURSO y SEMINARIO, y el vínculo IMPARTE se divide de manera correspondiente en IMPARTE y OFRECE. La figura 14.2(b) muestra un tipo de entidades OFERTA_CURSO que se refina para dar dos tipos de entidades y un vínculo entre ellos. La figura 14.3(a) muestra la primitiva de refinación ascendente de generar nuevos vínculos entre tipos de entidades. La refinación ascendente empleando generalización se ilustra en la figura 14.3(b), donde se "descubre" el nuevo concepto de DUEÑO_VEHÍCULO a partir de los tipos de entidades existentes PROFESORADO, ESTUDIANTE y PERSONAL; este proceso de generalización y la notación diagramática correspondiente se analizarán en el capítulo 21.

INTEGRACIÓN DE ESQUEMAS (VISTAS): En el caso de bases de datos grandes que se espera tendrán muchos usuarios y aplicaciones, puede usarse el enfoque de integración de vistas, diseñando esquemas individuales y luego combinándolos. Ya que es posible tener vistas individuales relativamente pequeñas, el diseño de los esquemas se simplifica. Sin embargo, se requiere una metodología para integrar las vistas en un esquema de base de datos global. La integración de esquemas puede dividirse en las siguientes subtareas:

1. Identificación de correspondencias y conflictos entre los esquemas: Dado que los esquemas se designan individualmente, es necesario especificar elementos en los esquemas que representen el mismo concepto del mundo real. Estas correspondencias deben identificarse antes de poder efectuar la integración. Durante este proceso, es posible que se descubran varios tipos de conflictos entre los esquemas:

a. Conflictos de nombres: Éstos son de dos tipos: sinónimos y homónimos. Un **sinónimo** ocurre cuando dos esquemas usan diferentes nombres para describir el mismo concepto; por ejemplo, un tipo de entidades COMPRADOR en un esquema puede describir el mismo concepto que un tipo de entidades CLIENTE en otro. Un **homónimo** ocurre cuando dos esquemas usan el mismo nombre para describir diferentes conceptos; por ejemplo un tipo de entidades COMPONENTE puede describir componentes de computador en un esquema y componentes de muebles en otro.
b. Conflictos de tipos: El mismo concepto puede representarse en dos esquemas mediante elementos de modelado diferentes. Por ejemplo, el concepto de DEPARTAMENTO puede ser un tipo de entidades en un esquema y un atributo en otro.
c. Conflictos de dominio (conjunto de valores): Un atributo puede tener diferentes dominios en dos esquemas. Por ejemplo NSS puede declararse como entero en un esquema y como cadena de caracteres en el otro. Podría haber conflicto de unidades de medida si un esquema representa PESO en libras y el otro en kilogramos.

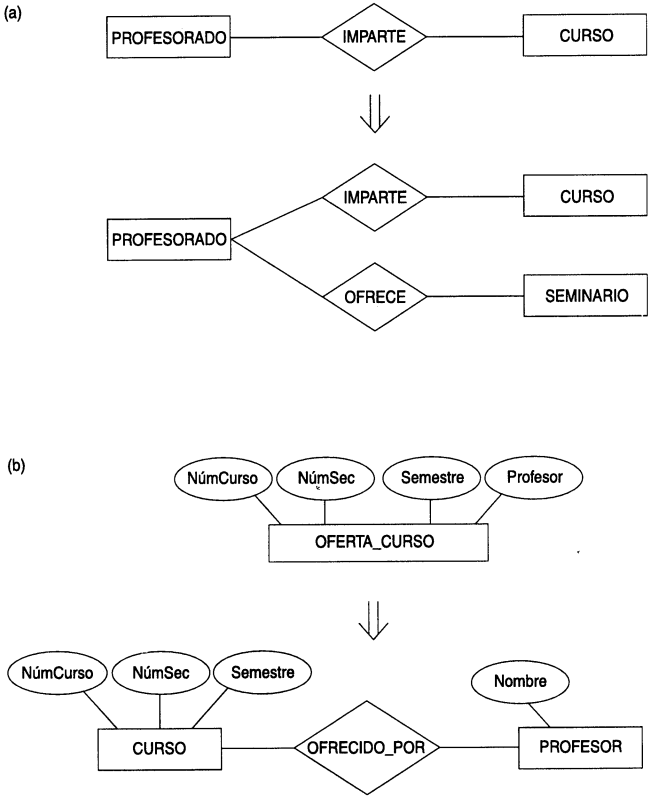


Figura 14.2 Ejemplos de refinación descendente. (a) Generación de un nuevo tipo de entidades. (b) Descomposición de un tipo de entidades en dos tipos de entidades y un vínculo.

d. Conflictos entre restricciones: Dos esquemas pueden imponer diferentes restricciones; por ejemplo, la clave de un mismo tipo de entidades puede ser diferente en cada esquema. Otro ejemplo implica diferentes restricciones estructurales sobre un vínculo como IMPARTE; un esquema puede representarlo como 1:*N* (un curso tiene un profesor) mientras el otro lo representa como M:*N* (un curso puede tener más de un profesor).

2. Modificación de las vistas para ajustarlas entre sí: Algunos esquemas se modifican de modo que se ajusten mejor a otros esquemas. Algunos de los conflictos identificados en el paso 1 se resuelven durante este paso.

3. Combinación de vistas: El esquema global se crea combinando los esquemas individuales. Los conceptos que se corresponden se representan sólo una vez en el esquema global, y se especifican las transformaciones de las vistas al esquema global.

4. Reestructuración: Como último paso opcional, el esquema global podría analizarse y reestructurarse para eliminar cualesquier redundancias o una complejidad innecesaria.

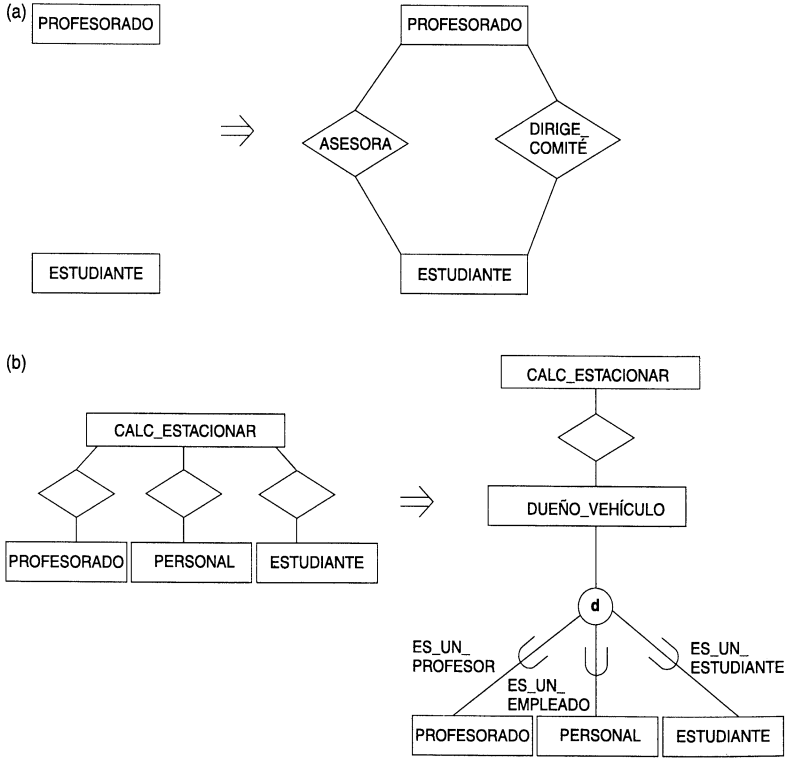


Figura 14.3 Ejemplos de refinación ascendente. (a) Descubrimiento y adición de nuevos vínculos. (b) Descubrimiento de un nuevo tipo de entidades generalizado, vinculándolo.

Algunas de estas ideas se ilustran con el ejemplo, más bien simple, que se presenta en las figuras 14.4 y 14.5. En la figura 14.4, combinamos dos vistas para crear una base de datos bibliográfica. Durante la identificación de correspondencias entre las dos vistas, descubrimos que INVESTIGADOR y AUTOR son sinónimos (en lo que a esta base de datos concierne), lo mismo que CONTRIBUIDO_POR y ESCRITO_POR. Además, decidimos modificar la vista 1 para que incluya un TEMA para ARTÍCULO, como se aprecia en la figura 14.4, *para ajustarla a* la vista 2. La figura 14.5 muestra el resultado de combinar la vista 1 modificada con la vista 2. Los vínculos PERTENECE_A y ESCRITO_POR se combinan, así como los tipos de entidades AUTOR y TEMA. También generalizamos los tipos de entidades ARTÍCULO y LIBRO para dar el tipo de entidades PUBLICACIÓN, con su atributo común, Título. El atributo Editor se aplica sólo al tipo de entidades LIBRO, en tanto que el atributo Tamaño y el tipo de vínculos PUBLICADO_EN se aplican sólo a ARTÍCULO.

Se han propuesto varias estrategias para el proceso de integración de vistas. Como se ilustra en la figura 14.6, éstas incluyen las siguientes:

1. Integración de escalera binaria: Primero se integran dos esquemas que sean muy similares. El esquema resultante se integrará entonces con otro esquema, y el proceso se repetirá hasta que todos los esquemas estén integrados. El ordenamiento de los esquemas para la integración puede basarse en alguna medida de la similitud de los esquemas. Esta estrategia es adecuada para la integración manual en virtud de su enfoque paso por paso.

2. Integración *n*-aria: Todas las vistas se integran mediante un procedimiento después de analizar y especificar sus correspondencias. Esta estrategia requiere herramientas computarizadas para problemas de diseño grandes.

3. Estrategia binaria balanceada: Primero se integran pares de esquemas; luego se aparean los esquemas resultantes para seguirlos integrando, y el procedimiento se repite hasta que se obtenga un esquema global final.

4. Estrategia mixta: En un principio, los esquemas se dividen en grupos con base en su similitud, y cada grupo se integra por separado. Los esquemas intermedios se agrupan otra vez y se integran, y así sucesivamente.

Fase 2b. Diseño de transacciones. El propósito de la fase 2b, que se realiza en paralelo con la fase 2a, es diseñar las características de las transacciones conocidas de la base de datos con independencia del SGBD. Cuando se está diseñando un sistema de base de datos, los diseñadores están conscientes de muchas aplicaciones conocidas (o **transacciones**) que se ejecutarán en la base de datos una vez que se implemente. Una parte importante del diseño de bases de datos es especificar las características funcionales de estas transacciones en una etapa temprana del proceso de diseño. Esto garantiza que el esquema de la base de datos incluirá toda la información requerida por dichas transacciones. Además, conocer la importancia relativa de las diversas transacciones y la frecuencia con que se espera invocarlas desempeña un papel crucial en el diseño físico de la base de datos (fase 5). Por lo regular, sólo se conocen algunas de las transacciones de la base de datos en el momento del diseño; una vez implementado el sistema se identificarán e implementarán continuamente nuevas transacciones. Sin embargo, es común que las transacciones más importantes se conozcan antes de la implementación del sistema, y deberán especificarse en una etapa temprana.

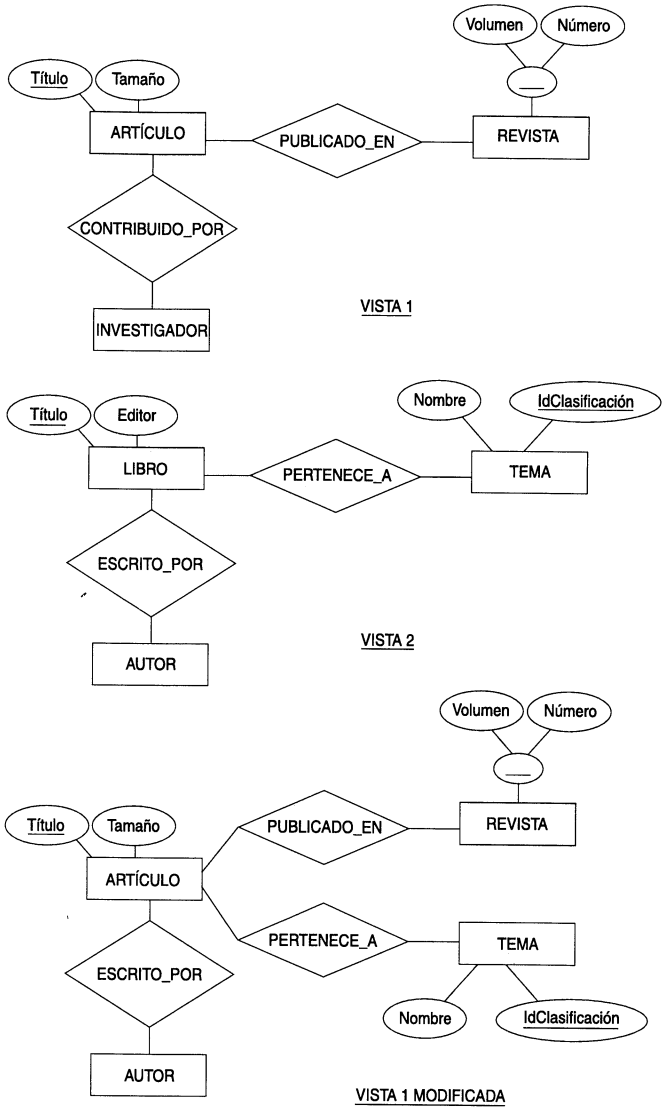


Figura 14.4 Modificación de las vistas para su ajuste antes de la integración.

Una técnica común para especificar transacciones en un nivel conceptual es identificar su comportamiento **de entrada/salida** y **funcional**. Al especificar los datos de entrada, los datos de salida y el flujo de control interno, los diseñadores pueden especificar una transacción en una forma conceptual independiente del sistema. Por lo regular, las transacciones

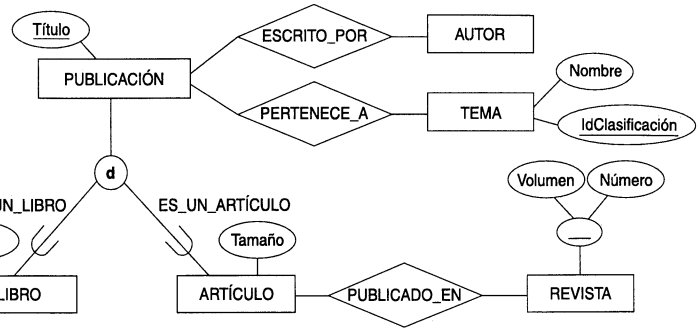


Figura 14.5 Esquema integrado después de combinar las vistas 1 y 2.

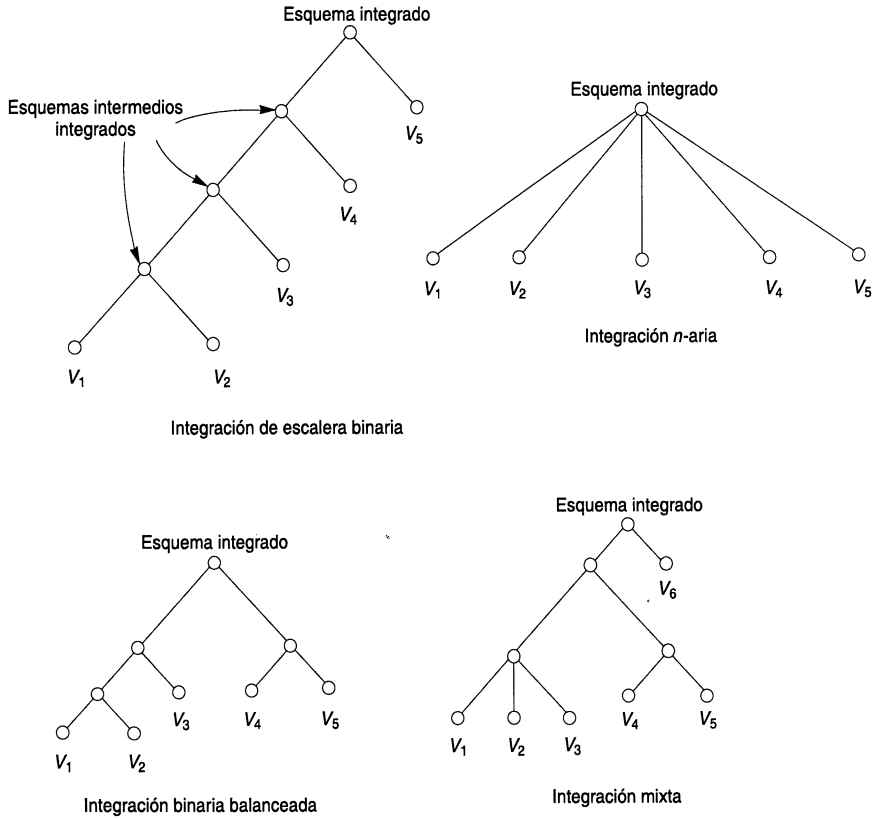


Figura 14.6 Diferentes estrategias para integrar vistas.

pueden agruparse en tres categorías: transacciones de obtención, transacciones de actualización y transacciones mixtas. Las **transacciones de obtención** sirven para obtener datos para exhibirlos en una pantalla o producir un informe. Las **transacciones de actualización** sirven para introducir datos nuevos o modificar datos que ya existen en la base de datos. Las **transacciones mixtas** se usan en aplicaciones más complejas que obtienen y actualizan datos. Por ejemplo, supongamos que estamos diseñando una base de datos de reservaciones en líneas aéreas. Un ejemplo de transacción de obtención sería listar todos los vuelos matutinos entre dos ciudades en una fecha dada. Un ejemplo de transacción de actualización consiste en reservar un asiento en un determinado vuelo. Una transacción mixta podría exhibir primero algunos datos, como mostrar la reservación de un cliente en un vuelo, y luego actualizar la base de datos, digamos cancelando la reservación al eliminarla.

Varias técnicas para especificar los requerimientos cuentan con una notación para especificar **procesos**, que en este contexto son operaciones más complejas que pueden consistir en varias transacciones. Otras propuestas para especificar transacciones incluyen TAXIS, GALILEO (véase la bibliografía) y GORDAS (véase la Sec. 21.5). El diseño de transacciones es tan importante como el diseño del esquema. Desafortunadamente, muchas metodologías de diseño actuales conceden mayor importancia a uno o al otro. Es mejor llevar a cabo las fases 2a y 2b en paralelo, mediante ciclos de retroalimentación para refinar, hasta lograr un diseño estable de esquema y transacciones.

14.2.3 *Fase 3: Elección del SGBD*

La elección del SGBD depende de varios factores, algunos de ellos técnicos, otros económicos y otros más relativos a las políticas de la organización. Los factores técnicos tienen que ver con la idoneidad del SGBD para la tarea en cuestión. Lo que debemos considerar aquí son el tipo de SGBD (relacional, de red, jerárquico, orientado a objetos o de otra clase), las estructuras de almacenamiento y caminos de acceso que maneja el SGBD, las interfaces de usuario y programador disponibles, los tipos de lenguajes de consulta de alto nivel, etc. Examinaremos estos factores técnicos en el apéndice C, cuando comparemos los modelos de datos. En esta sección nos concentraremos en los factores económicos y de organización que influyen en la elección del SGBD. Al escoger un SGBD, debemos considerar los siguientes costos:

1. Costo de adquisición del software: Éste es el gasto inicial que se hace al comprar el software, en el que se incluyen las opciones de lenguajes, diferentes interfaces como formas y pantallas, opciones de recuperación y respaldo, métodos de acceso especiales y documentación. Debe seleccionarse la versión correcta del SGBD para el sistema operativo específico que se tiene.
2. Costo de mantenimiento: Éste es el costo recurrente por concepto del servicio de mantenimiento del proveedor y de la actualización regular de la versión del SGBD.
3. Costo de adquisición de hardware: Tal vez se requiera más equipo, como memoria adicional, terminales, unidades de disco o almacenamiento especializado para el SGBD.
4. Costo de creación y conversión de la base de datos: Éste es el costo de crear el sistema de base de datos desde cero o bien de convertir un sistema existente al nuevo software de SGBD. En este último caso, se acostumbra operar el sistema existente en paralelo con el nuevo sistema hasta que todas las aplicaciones recién adquiridas estén perfectamente implementadas y probadas. Es difícil pronosticar este costo, y con frecuencia se subestima.
5. Costo de personal: Cuando una organización adquiere por primera vez un software de SGBD a menudo emprende un proceso de reorganización de su departamento de procesamiento de datos. En casi todas las compañías que adoptan un SGBD se deben crear puestos nuevos para el administrador de bases de datos (DBA) y para su personal.
6. Costo de capacitación: Como los SGBD suelen ser sistemas complejos, casi siempre es preciso capacitar al personal para su uso y programación.
7. Costo de operación: El costo de la operación continua del sistema de base de datos no suele incluirse en la evaluación de las alternativas porque es independiente del SGBD que se seleccione.

No es tan fácil medir y cuantificar los beneficios de adquirir un SGBD. Un SGBD tiene varias ventajas intangibles respecto a los sistemas de archivos tradicionales, como serían la facilidad de uso, la mayor disponibilidad de los datos y un acceso más rápido a la información. Entre los beneficios más tangibles están la reducción en el costo de crear aplicaciones, la menor redundancia de los datos y el mejor control y seguridad. Con base en un análisis de costo-beneficios, una organización tiene que decidir cuándo cambiar a un SGBD. En general, este paso depende de los siguientes factores:

- Complejidad de los datos: Cuanto más complejas sean las interrelaciones de los datos, mayor será la necesidad de contar con un SGBD.
- Compartimiento entre aplicaciones: Cuanto más se compartan los datos entre las aplicaciones, mayor redundancia habrá entre los archivos, así que será más necesario un SGBD.
- Evolución o crecimiento dinámico de los datos: Si hay cambios constantes en los datos, es más fácil manejarlos con un SGBD que con un sistema de archivos.
- Frecuencia de solicitudes de datos *ad hoc*: Los sistemas de archivos no son en absoluto apropiados para la obtención *ad hoc* de datos.
- Volumen de los datos y necesidad de control: Grandes volúmenes de datos y la necesidad de controlarlos a veces hacen obligatorio un SGBD.

Por último, hay varios factores económicos y de organización que influyen en la elección de un SGBD en vez de otro:

1. Estructura de los datos: Si los datos que se almacenarán en la base de datos tienen una estructura jerárquica, deberá considerarse la adquisición de un SGBD de tipo jerárquico. Si los datos tienen muchos vínculos, quizá sea más apropiado un sistema de red o relacional. La tecnología relacional cada vez es más popular. En el caso de estructuras y tipos de datos complejos, quizá sean adecuados los sistemas orientados a objetos.
2. Familiaridad del personal con el sistema: Si el personal de programación en la organización ya conoce un SGBD determinado, éste puede ser más recomendable por reducir los costos de capacitación y el tiempo de aprendizaje.

3. Disponibilidad de servicios del proveedor: Es deseable que haya una oficina de servicio del proveedor en los alrededores para ayudar a resolver cualesquier problemas que se presenten con el sistema. Cambiar de un sistema que no es de SGBD a uno que sí lo es casi siempre es un paso importante y requiere mucha ayuda del proveedor al principio.

Antes de adquirir un SGBD, la organización debe tener en cuenta la configuración de hardware/software que éste requiera para su ejecución, así como su transportabilidad entre los diferentes tipos de hardware. Muchos SGBD comerciales ya cuentan con versiones que se ejecutan en muchas configuraciones de hardware/software (o **plataformas**). También debe considerarse la necesidad de aplicaciones para respaldo, recuperación, rendimiento, integridad y seguridad. Hoy por hoy, muchos SGBD se están diseñando como *soluciones totales* a las necesidades de procesamiento de información y gestión de recursos de información que tienen las organizaciones. En su mayoría, los proveedores de SGBD están combinando sus productos con las siguientes opciones o características integradas, que a menudo califican como 4GL (lenguajes de cuarta generación):

- Editores de texto y examinadores.
- Generadores de informes y utilerías para listados.
- Software de comunicación (a menudo llamado supervisor de teleproceso).
- Características de introducción y exhibición de datos, como son formas, pantallas y menús con funciones de edición automática.
- Herramientas de diseño gráficas.

En algunos casos tal vez no resulte apropiado usar un SGBD, sino más bien crear programas propios para las aplicaciones. Esto puede ocurrir si las aplicaciones están muy bien definidas y *todas* se conocen por anticipado. En una situación así, un sistema diseñado a la medida dentro de la organización puede ser apropiado para implementar las aplicaciones conocidas de la manera más eficiente. Sin embargo, en la mayoría de los casos surgen nuevas aplicaciones no previstas a la hora del diseño *después* de la implementación del sistema. Ésta es precisamente la razón por la cual los SGBD se han vuelto tan populares: facilitan la incorporación de nuevas aplicaciones sin tener que hacer cambios importantes al sistema existente.

14.2.4 *Fase 4: Transformación al modelo de datos (diseño lógico de la base de datos)*

La siguiente fase del diseño de la base de datos consiste en crear un esquema conceptual y esquemas externos en el modelo de datos del SGBD elegido. Esto se logra transformando los esquemas conceptual y externos producidos en la fase 2a del modelo de datos de alto nivel al modelo de datos del SGBD. La transformación puede establecerse en dos etapas:

1. Transformación independiente del sistema: En este paso, la transformación al modelo de datos del SGBD no considera las características específicas o casos especiales que se aplican a la forma como el SGBD implementa el modelo de datos. Analizamos la transformación de un esquema ER a un esquema relacional, orientado a objetos, jerárquico o de red, de manera independiente del SGBD, en las secciones 6.8, 22.8, 10.4 y 11.5, respectivamente.

2. Adaptación de los esquemas a un SGBD específico: Los diferentes SGBD implementan un modelo de datos con características y restricciones de modelado específicas. Tal vez sea preciso ajustar los esquemas obtenidos en el paso 1 para adaptarlos a las características de implementación específicas de un modelo de datos en el SGBD seleccionado.

El resultado de esta fase debe consistir en enunciados DDL escritos en el lenguaje del SGBD elegido que especifiquen los esquemas a nivel conceptual y externo del sistema de base de datos. Sin embargo, si los enunciados DDL contienen algunos parámetros de diseño físico, la especificación completa en DDL deberá esperar hasta que se haya concluido la fase de diseño físico. Muchas herramientas CASE (*computer assisted software engineering*: ingeniería de software asistida por computador) de diseño automatizado (véase la Sec. 14.4) pueden generar DDL para sistemas comerciales a partir de un diseño de esquema conceptual.

14.2.5 *Fase 5: Diseño físico de la base de datos*

El diseño físico de la base de datos es el proceso de elegir estructuras de almacenamiento y caminos de acceso específicos para que los archivos de la base de datos tengan un buen rendimiento con las diversas aplicaciones de la base de datos. Cada SGBD ofrece varias opciones de organización de archivos y caminos de acceso, entre ellas diversos tipos de indización, agrupamiento de registros relacionados en bloques de disco, enlace de registros relacionados mediante apuntadores y varios tipos de técnicas de dispersión. Una vez seleccionado un SGBD específico, el proceso de diseño físico se reduce a elegir las estructuras más apropiadas para los archivos de la base de datos entre las opciones que ofrece ese SGBD. En esta sección presentaremos pautas para las decisiones de diseño físico en diversos tipos de SGBD. A menudo se utilizan los siguientes criterios para guiar la elección de las opciones de diseño físico:

1. Tiempo de respuesta: Éste es el tiempo que transcurre entre la introducción de una transacción de base de datos para ser ejecutada y la obtención de una respuesta. Un aspecto que influye mucho sobre el tiempo de respuesta y que está bajo el control del SGBD es el tiempo de acceso a la base de datos para obtener los elementos de información a los que hace referencia la transacción. El tiempo de respuesta también depende de factores que no puede controlar el SGBD, como son la carga del sistema, la planificación de tareas del sistema operativo o los retrasos de comunicación.
2. Aprovechamiento del espacio: Esto se refiere a la cantidad de espacio de almacenamiento que ocupan los archivos de la base de datos y sus estructuras de acceso.
3. Productividad de las transacciones: Éste es el número promedio de transacciones que el sistema de base de datos puede procesar por minuto; es un parámetro crítico de los sistemas de transacciones como los que se usan para las reservaciones en líneas aéreas o el servicio en los bancos. La productividad de transacciones debe medirse en las condiciones pico del sistema.

Por lo regular se especifican límites promedio y del peor de los casos para los parámetros anteriores como parte de los requerimientos de rendimiento del sistema. Se utilizan técnicas analíticas o experimentales, como la creación de prototipos y la simulación, para estimar los valores promedio y del peor de los casos suponiendo diferentes decisiones de diseño físico, a fin de determinar si satisfacen los requerimientos de rendimiento especificados.

El rendimiento depende del tamaño y del número de registros que contienen los archivos; por ello, debemos estimar estos parámetros para cada archivo. Por añadidura, debemos estimar los patrones de actualización y obtención de datos para el archivo colectivamente a partir de todas las transacciones. Es recomendable construir caminos de acceso primarios e índices secundarios para los atributos con que se seleccionan los registros. También deben tenerse en cuenta durante la fase de diseño físico las estimaciones del crecimiento de los archivos, sea en el tamaño de los registros debido a la adición de nuevos atributos o en el número de registros.

El resultado de la fase de diseño físico es una determinación *inicial* de las estructuras de almacenamiento y los caminos de acceso para los archivos de la base de datos. Casi siempre es necesario **afinar** el diseño con base en su rendimiento observado después de la implementación del sistema. La mayor parte de los sistemas cuenta con una utilería de supervisión que reúne datos estadísticos de rendimiento, los cuales se conservan en el catálogo del sistema o en el diccionario de datos para su análisis posterior. Estos datos incluyen el número de invocaciones de transacciones o consultas predefinidas, actividades de entrada/salida de cada archivo, conteo de páginas de archivos o registros de índice, y frecuencia de utilización de índices. Al cambiar los requerimientos del sistema de base de datos, suele hacerse necesaria la reorganización de algunos archivos mediante la construcción de índices nuevos o la modificación de los métodos de acceso primarios. En la sección 14.3 estudiaremos cuestiones de diseño físico relacionadas con los diferentes tipos de SGBD.

14.2.6 *Fase 6: Implementación del sistema de base de datos*

Una vez completados los diseños lógico y físico, podemos implementar el sistema de base de datos. Se compilan los enunciados escritos en el DDL (lenguaje de definición de datos) y en el SDL (*storage definition language*: lenguaje de definición de almacenamiento) del SGBD seleccionado, y con ellos se crean los esquemas de la base de datos y sus archivos (vacíos). En seguida ya puede **cargarse** (poblarse) la base de datos. Si los datos tienen que convertirse de un sistema computarizado antiguo, tal vez se requieran **rutinas de conversión** para modificar el formato de los datos y así poderlos almacenar en la nueva base de datos.

En esta etapa, los programadores de aplicaciones deben implementar las transacciones de la base de datos. Se examinan las especificaciones conceptuales de las transacciones, y se escribe y se prueba el código de programa correspondiente con órdenes de DML (lenguaje de manipulación de datos) incorporadas. Una vez que las transacciones estén listas y los datos se hayan almacenado en la base de datos, la fase de diseño e implementación habrá terminado y se iniciará la fase de operación del sistema.

14.3 Pautas para el diseño físico de bases de datos*

En esta sección comenzaremos por estudiar los factores de diseño físico que afectan el rendimiento de las aplicaciones y de las transacciones; luego comentaremos las pautas específicas para los SGBD relacionales, de red y jerárquicos.

14.3.1 *Factores que influyen en el diseño físico de la base de datos*

El diseño físico es una actividad en la que el objetivo no sólo es lograr la estructuración apropiada de los datos en el almacenamiento, sino hacerlo de manera tal que garantice un buen

rendimiento. Para un esquema conceptual dado, existen muchas alternativas de diseño físico en un SGBD en particular. No es posible tomar decisiones de diseño físico ni realizar análisis de rendimiento que tengan sentido antes de conocer las consultas, transacciones y aplicaciones que se piensa ejecutar en la base de datos. Debemos analizar estas aplicaciones, sus frecuencias de invocación esperadas, cualesquier restricciones de tiempo que haya sobre su ejecución y la frecuencia esperada de las operaciones de actualización. En seguida examinaremos cada uno de estos factores.

Análisis de las consultas y transacciones de la base de datos. Antes de emprender el diseño físico de la base de datos, debemos tener una idea bastante precisa del uso que se le piensa dar, definiendo a alto nivel las consultas y transacciones que se espera ejecutar en ella. Para cada *consulta*, deberemos especificar lo siguiente:

1. Los archivos a los que tendrá acceso la consulta.
2. Los campos sobre los cuales se especificarán cualesquier condiciones de selección incluidas en la consulta.
3. Los campos sobre los cuales se especificarán cualesquier condiciones de reunión o condiciones para enlazar múltiples tipos de registros incluidas en la consulta.
4. Los campos cuyos valores obtendrá la consulta.

Los campos a los que se refieren los apartados 2 y 3 son candidatos para la definición de estructuras de acceso. Para cada transacción u operación de actualización deberemos especificar lo siguiente:

1. Los archivos que se actualizarán.
2. El tipo de operación de actualización en cada archivo (insertar, modificar o eliminar).
3. Los campos sobre los que se especificarán cualesquier condiciones de selección para una operación de eliminación o modificación.
4. Los campos cuyos valores alterará una operación de modificación.

Una vez más, los campos mencionados en el apartado 3 son candidatos para estructuras de acceso. Por otro lado, los campos indicados en el apartado 4 son candidatos para evitar estructuras de acceso, ya que su modificación requeriría la actualización de dichas estructuras.

Análisis de la frecuencia esperada de invocación de consultas y transacciones. Además de identificar las características de las consultas y transacciones esperadas, debemos considerar sus tasas (o velocidades) esperadas de invocación. Esta información de frecuencias, junto con aquélla recabada sobre los campos para cada consulta y transacción, servirá para compilar una lista colectiva de frecuencia de uso esperada para todas las consultas y transacciones. Esto se expresa como la frecuencia esperada de uso de cada campo de cada archivo como campo de selección o campo de reunión, considerando todas las consultas y transacciones. En general, cuando el volumen de procesamiento es elevado se aplica la regla informal «80-20». Esta regla establece que apenas el 20% de las consultas y transacciones dan cuenta de aproximadamente el 80% del procesamiento. Por tanto, en situaciones prácticas casi nunca es necesario recabar datos estadísticos y tasas de invocación exhaustivos para todas las consultas y transacciones; basta con determinar alrededor del 20% de ellas que comprende las más importantes.

Análisis de las restricciones de tiempo sobre las consultas y transacciones. Es posible que algunas consultas y transacciones posean restricciones de rendimiento muy exigentes. Por ejemplo, una cierta transacción puede tener la restricción de que debe terminar antes de 5 segundos en el 95% de las veces que se invoque, y que nunca debe tardar más de 20 segundos. Semejantes consideraciones de rendimiento pueden servir para asignar prioridades adicionales a los campos que son candidatos para caminos de acceso. Los campos de selección utilizados por las consultas y transacciones que tienen restricciones de tiempo se convertirán en candidatos de mayor prioridad para las estructuras de acceso.

Análisis de las frecuencias esperadas de las operaciones de actualización. Se debe especificar el mínimo posible de caminos de acceso para los archivos que se actualizan con frecuencia, porque la actualización de los caminos de acceso mismos hace más lentas las operaciones.

Una vez compilada la información anterior, podemos abordar las decisiones de diseño físico, que consisten principalmente en decidir qué estructuras de almacenamiento y caminos de acceso tendrán los archivos de la base de datos. Rara vez se conocen todas las consultas y transacciones de la base de datos en el momento en que se hace el diseño físico. A menudo, después de implementarse el sistema de base de datos surgen nuevas aplicaciones, lo que obliga a especificar nuevas consultas y transacciones. En tales casos, se hará necesario modificar algunas de las decisiones de diseño físico para poder incorporar las nuevas aplicaciones al sistema. Esto se conoce como **afinación** del diseño físico. Si algunas de las transacciones o consultas con restricciones de tiempo de respuesta no cumplen con los tiempos especificados, se requerirán modificaciones del diseño físico original a fin de mejorar la eficiencia de esas transacciones.

14.3.2 *Pautas de diseño físico de bases de datos para sistemas relacionales*

En su mayoría, los sistemas relacionales representan cada relación base como un archivo de base de datos físico. Las opciones de caminos de acceso incluyen la especificación del tipo de archivo para cada relación y los atributos sobre los cuales habrán de definirse índices. Uno de los índices de cada archivo puede ser primario o de agrupamiento. Además, hay varias técnicas con que se aceleran las operaciones de EQUIRREUNIÓN o REUNIÓN NATURAL, que son muy frecuentes. Analizaremos primero estas técnicas y luego trataremos la elección de organizaciones de archivos y de índices.

Técnicas para acelerar las operaciones de EQUIRREUNIÓN o de REUNIÓN NATURAL. Algunos sistemas relacionales ofrecen la opción de almacenar dos relaciones con un vínculo 1:*N*, representado por una *clave externa*, como un solo archivo jerárquico de dos niveles, donde se almacena cada registro del lado 1 (clave primaria) seguido de los registros del lado *N* (con claves externas coincidentes). Este tipo de estructura de almacenamiento hace muy eficiente la REUNIÓN entre los dos archivos de este vínculo 1:*N*. Por ejemplo, consideremos el esquema de base de datos relacional de la figura 6.5, y supongamos que cada relación se implementa como un archivo. Si es preciso realizar la EQUIRREUNIÓN con la condición EMPLEADO.NSS = TRABAJA_EN.NSSE de la manera más eficiente posible, podemos usar un archivo mixto de

los registros EMPLEADO y TRABAJA_EN. Cada registro EMPLEADO se almacenará seguido de los registros TRABAJA_EN que tengan el mismo valor de NSS. También es posible *agrupar* los registros de TRABAJA_EN por NSSE sin combinar los archivos.

Otra opción consiste en **desnormalizar** el esquema lógico de la base de datos cuando diseñemos los archivos físicos. Esto se hace para colocar los atributos que se usan con frecuencia en una consulta en los mismos registros de archivo que otros atributos que intervienen en la consulta. Hacemos esto **repitiendo** (o **duplicando**) los atributos en el archivo en que se necesitan. Debemos compensar esta desnormalización exigiendo a las transacciones de actualización mantener la consistencia de los valores de los atributos duplicados. Por ejemplo, supongamos que la misma consulta que requiere la operación de REUNIÓN anterior también precisa la obtención del atributo NOMBREPR del registro PROYECTO en el que PROYECTO.NÚMEROP = TRABAJA_EN.NÚMP. En tal caso podríamos duplicar el atributo NOMBREPR en los registros TRABAJA_EN para que no tengamos que efectuar esta última reunión. Esta técnica puede llevarse un paso más lejos almacenando físicamente un archivo que sea el resultado de la REUNIÓN de dos archivos, aunque esta desnormalización extrema debe adoptarse con muchísimo cuidado, ya que pueden presentarse todos los tipos de anomalías de actualización que vimos en el capítulo 12, mismas que es preciso resolver explícitamente siempre que se aplican actualizaciones al archivo.

Pautas para la organización de los archivos y la selección de índices. Una opción preferida para organizar los archivos individuales en un sistema relacional consiste en mantener no ordenados los registros de los archivos y crear tantos índices secundarios como sea preciso. Los atributos que se usan a menudo en las condiciones de selección y de reunión son candidatos para ser índices secundarios.

Otra opción consiste en especificar un atributo de ordenación para el archivo especificando un índice primario o de agrupamiento. El atributo más utilizado para las operaciones de reunión deberá ser el que se seleccione para ordenar o agrupar los registros, ya que esto hace más eficiente la operación de reunión (véase el Cap. 16). Si a menudo se tiene acceso a los registros en orden según un atributo, ésta es otra indicación de que debemos elegirlo para ordenar los registros del archivo. Sólo uno de los atributos de cada archivo puede escogerse para tal ordenamiento físico, con un índice primario (si el atributo es una clave) o un índice de agrupamiento (si no lo es) correspondiente. En muchos sistemas relacionales se usan las palabras reservadas UNIQUE (único) para especificar una clave y CLUSTER para especificar un índice de agrupamiento.

Elección de dispersión. Algunos sistemas relacionales ofrecen además la opción de especificar un atributo como clave de dispersión en vez de un atributo de ordenamiento para una relación. Si un atributo clave se va a usar principalmente para seleccionar por igualdad y para operaciones de reunión, pero no para tener acceso a los registros en orden, podemos escoger un archivo disperso en vez de uno ordenado. Otro criterio para elegir un archivo disperso es que se conozca el tamaño del archivo y no se esperen crecimientos o reducciones significativos. Si el tamaño del archivo cambia y el SGBD relacional cuenta con algún esquema de dispersión dinámica (véase el Cap. 4), podemos escoger la dispersión dinámica para ese archivo.

Podemos resumir las pautas para elegir una organización física para un archivo relacional individual como sigue:

1. Escoger como atributo de ordenación para el archivo aquel que se use con frecuencia para obtener los registros en orden o que se use más a menudo en operaciones de reunión con ese archivo. Crear un índice primario (si el atributo es una clave) o un índice de agrupamiento (si no lo es) según ese atributo. Éste es el *camino de acceso primario* a los registros del archivo. Si ningún atributo satisface estos criterios, usar un archivo no ordenado.
2. Para cada atributo (diferente del atributo de ordenación) que se use con frecuencia en operaciones de selección o de reunión, especificar un índice secundario que sirva como *camino de acceso secundario* a los registros del archivo.
3. Si el archivo se va a actualizar muy a menudo con inserción y eliminación de registros, reducir al mínimo posible el número de índices del archivo.
4. Si un atributo se usa con frecuencia para la selección por igualdad o para operaciones de reunión pero no para leer los registros en orden, puede usarse un archivo disperso. Se puede usar dispersión estática si el tamaño del archivo no cambia mucho, pero se requerirá dispersión dinámica en el caso de archivos que crezcan rápidamente o cuyo tamaño fluctúe. Se pueden construir índices secundarios sobre otros atributos del archivo disperso.

Desde luego, la elección de una de estas opciones está limitada por las estructuras de almacenamiento y caminos de acceso disponibles en el SGBD relacional que se haya escogido.

14.3.3 *Pautas de diseño físico de bases de datos para sistemas de red*

Existen varias opciones de diseño físico importantes para una base de datos de red. La primera implica decidir cómo implementar cada tipo de conjuntos. Otra elección, similar a la tocante a la desnormalización en las bases de datos relacionales, implica decidir si uno o más campos de un tipo de registros propietario deben repetirse o no en un tipo de registros miembro por razones de eficiencia. Además, algunos SGBD de red permiten definir claves de dispersión o índices para un tipo de registros. Éstos, junto con los tipos de conjuntos propiedad del sistema, proveen puntos de entrada a la base de datos, que permiten después *navegar por la base de datos* siguiendo los apuntadores de los conjuntos (**procesamiento de conjuntos**). Por último, es preciso tomar decisiones respecto a los campos de ordenamiento de los tipos de registros o de los registros miembro dentro de un tipo de conjuntos.

Pautas para elegir entre las opciones de implementación de conjuntos. Hay muchas opciones para implementar un tipo de conjuntos. Repasaremos esas opciones, que estudiamos en el capítulo 10, y presentaremos pautas para decidir cuál opción utilizar, a saber:

1. Implementar un tipo de conjuntos por contigüidad física: En algunos SGBD de red, los registros miembro pueden almacenarse físicamente después del registro propietario; esta opción se denomina implementación de conjuntos por **contigüidad física.** Si un tipo de registros es miembro de varios tipos de conjuntos, *sólo uno* de éstos podrá escogerse para implementación por contigüidad física. El acceso a los registros miembro de un ejemplar de conjunto tiene eficiencia máxima con esta opción, así que los tipos de conjuntos que se usen con mayor frecuencia para tener acceso a *todos los registros miembro* de un propietario son candidatos para esta opción de implementación.

2. Implementar un tipo de conjuntos con arreglos de apuntadores: Otra opción consiste en almacenar un arreglo de apuntadores a los registros miembro junto con el registro propietario. Si un *registro miembro individual* se selecciona por su posición en el conjunto (como FIRST (primero), LAST (último) o *i*-ésimo) y a menudo desde el propietario, ésta será una buena opción.
3. Emplear diferentes opciones para la implementación de apuntadores: Casi todos los conjuntos se implementan con apuntadores y listas enlazadas. Podemos tener un solo apuntador **al siguiente** o apuntadores **al siguiente** y **al anterior** en los registros miembro de un conjunto. Con cualquiera de estas dos opciones, podemos tener además un apuntador **al propietario** en cada registro miembro. Si el principal acceso a los miembros del conjunto se tiene mediante FIND NEXT (buscar el siguiente), bastará con el apuntador al siguiente. Si se utiliza con frecuencia FIND PRIOR (buscar previo), se deberá incluir un apuntador al anterior. Por último, si es común tener acceso al propietario desde un registro miembro usando FIND OWNER, conviene incluir un apuntador al propietario. Esta última opción es útil para tipos de registros que participan como miembros en muchos tipos de conjuntos; por ejemplo, los tipos de registros de enlace para vínculos M:*N*, como el tipo de registros TRABAJA_EN de la figura 10.9. Con esta opción, un programa puede obtener los registros miembro usando un tipo de conjuntos —digamos E_TRABAJAEN— y luego encontrar directamente el registro propietario mediante el apuntador al propietario del otro tipo de conjuntos (P_TRABAJAEN).

En una base de datos de red, los equivalentes de la mayoría de las operaciones de EQUIRREUNIÓN relacionales se *preespecifican* como tipos de conjuntos, como puede verse si comparamos el esquema de la base de datos COMPAÑÍA en el modelo relacional (Fig. 6.5) y en el modelo de red (Fig. 10.9). Por ejemplo, la condición de reunión EMPLEADO.NSS = TRABAJA_EN.NSSE en el esquema relacional se representa con el tipo de conjuntos E_TRABAJAEN. De la misma manera, la condición de reunión PROYECTO.NÚMEROP = TRABAJA_EN.NÚMP se representa con el tipo de conjuntos P_TRABAJAEN. Así pues, las reuniones que se ejecutan con frecuencia en el modelo relacional corresponden a conjuntos que se recorren con frecuencia en el modelo de red. Tales tipos de conjuntos son candidatos para una implementación eficiente por contigüidad física.

Desnormalización por razones de eficiencia o de restricciones estructurales. Quizá deseemos duplicar algunos de los campos de un tipo de registros *propietario* en el tipo de registros *miembro* de un tipo de conjuntos, por las siguientes razones:

- Si se repite un campo (clave) NO DUPLICATES ALLOWED del propietario, con él puede especificarse una restricción estructural para un tipo de conjuntos MANUAL, o SET SELECTION IS AUTOMATIC para un tipo de conjuntos AUTOMATIC. Los valores de estos campos repetidos pueden leerse directamente desde el registro miembro, *sin que el* SGBD *tenga que localizar y leer el registro propietario*, reduciéndose así el tiempo de acceso, sobre todo si el registro miembro no tiene apuntador al propietario.
- Pueden repetirse también otros campos, no clave, en un miembro por razones de eficiencia. Sin embargo, esta repetición significa que el programa de actualización deberá propagar a todos los registros miembro las actualizaciones de un campo del propietario que se repita en los miembros, a fin de conservar la consistencia, cosa que hará más lentas las operaciones de actualización.

Opciones de acceso a registros. El modelo de red requiere **puntos de entrada** a la base de datos para iniciar las búsquedas de registros por recorrido o navegación. Estos puntos se establecen mediante tipos de conjuntos propiedad del sistema o especificando una estructura de acceso para un tipo de registros. La acción por omisión consiste en efectuar una búsqueda lineal en un **área**: un concepto de SGBD que representa una partición lógica de la base de datos asignada a un área físicamente contigua en el disco. Otros **modos de localización** (LOCATION MODE) de tipos de registros en el informe DBTG original, que todavía se siguen en muchos SGBD de red comerciales, son los siguientes:

- *CALC* — Los registros del tipo de registros se dispersan según un campo especificado del tipo que se define como CALC KEY (clave de cálculo). Un registro se obtiene directamente por el valor de su CALC KEY.
- *VIA SET* — Esto hace que los registros miembro se almacenen cerca del propietario; no se dispone de acceso directo a los registros.
- *DIRECT* — El programa de aplicación sugiere una página física en la cual o cerca de la cual se deberá almacenar el registro. La dirección real de su ubicación (en forma de apuntador al registro) se devuelve en una clave de base de datos (**DBKEY**). El concepto de clave de base de datos fue propuesto en el informe DBTG original como un mecanismo de eficiencia, pero se eliminó de informes posteriores.

La opción directa se utiliza cuando un programa conserva un apuntador al registro y luego obtiene con él el registro directamente. El concepto de área permite al diseñador especificar que los registros de ciertos tipos se coloquen *físicamente cercanos entre sí* en el disco, tal vez en el mismo cilindro. Esto acelera considerablemente el acceso a varios de estos registros, sobre todo cuando están relacionados a través de conjuntos. La especificación de áreas es una importante decisión de diseño físico en los sistemas que manejan este concepto.

Selección de conjuntos propiedad del sistema y de ordenamiento de registros. Los conjuntos propiedad del sistema se definen para procesar todos los registros de un tipo en algún orden deseado, casi siempre para aplicaciones de generación de informes. Los registros de un conjunto propiedad del sistema pueden ordenarse según los valores de un campo especificado en la cláusula ORDER. Es recomendable especificar conjuntos propiedad del sistema si se piensa usar los registros de un tipo como "puntos de entrada" para después localizar los registros relacionados con ellos a través de otros conjuntos. Por ejemplo, si a menudo preparamos informes que imprimen información sobre los departamentos y presentan información de los empleados y proyectos de esos departamentos, podemos usar un conjunto propiedad del sistema para tener acceso a los registros DEPARTAMENTO de la figura 10.9. Después podemos obtener los registros EMPLEADO y PROYECTO relacionados a través de los conjuntos PERTENECE_A, DIRIGE y CONTROLA. Si por lo regular tenemos acceso a los departamentos en orden por número de departamento, podemos ordenar el conjunto propiedad del sistema según el campo NÚMD.

14.3.4 *Pautas de diseño físico de bases de datos para sistemas jerárquicos*

Las principales decisiones en torno al diseño físico de bases de datos en los sistemas jerárquicos están íntimamente relacionadas con el diseño lógico debido a las múltiples opciones con que contamos para especificar jerarquías en el mismo esquema conceptual. Sin

embargo, existen varias opciones adicionales, como la elección de campos de dispersión o indización para el tipo de registros raíz o la elección de indización «secundaria» para los tipos de registros no raíz y la implementación de los vínculos padre-hijo virtuales. El acceso a los datos de una base de datos jerárquica está restringido por la estructura jerárquica y casi siempre se efectúa localizando primero el registro raíz. Dentro de un árbol de ocurrencia, la búsqueda se realiza ya sea secuencialmente por los registros del árbol o siguiendo ciertos esquemas de apuntadores. Los registros raíz pueden tener acceso indizado o disperso según ciertos campos para localizar con eficiencia el árbol de ocurrencia requerido. Nos enfrentamos a las siguientes decisiones que afectan el rendimiento de la base de datos:

1. Elección de tipos de registros raíz de las jerarquías: Los tipos de registros raíz son puntos de entrada a la base de datos, porque es posible tener acceso a todos los registros descendientes desde la raíz. Por añadidura, es fácil especificar caminos de acceso, como la dispersión y los índices, con base en la raíz. Los tipos de registros que se usan con frecuencia para iniciar una obtención son buenos candidatos para ser raíces de jerarquías.
2. Opciones de implementación de vínculos padre-hijo (VPH): La implementación más común de un VPH es como archivo jerárquico, que utiliza contigüidad física y almacena los registros como secuencia jerárquica (recorrido en preorden), como se vio en el capítulo 11. Sin embargo, es posible añadir apuntadores para facilitar la localización de registros descendientes de un cierto tipo (llamados *índices secundarios* en IMS). De manera similar, también es posible añadir apuntadores para facilitar la localización de un registro antecesor (llamados *apuntadores a padres físicos* en IMS).
3. Elección de registros apuntadores: La opción del tipo de registros apuntador minimiza la redundancia a expensas de tener que definir un vínculo padre-hijo virtual (VPHV). Es importante la decisión de diseño de elegir entre esta opción y la repetición de registros en una jerarquía. La primera opción minimiza la redundancia, pero la segunda ofrece una obtención más eficiente a expensas de complicar enormemente el proceso de actualización.
4. Diferentes opciones para la implementación de vínculos padre-hijo virtuales: La mayoría de los VPHV se implementan mediante apuntadores en los registros hijo, lo que facilita la localización del registro padre desde el registro hijo. Esto es similar al apuntador al propietario en el modelo de red. Si se desea proveer acceso desde un padre virtual a su primer hijo virtual, puede usarse un apuntador (llamado *apuntador a hijo virtual* en IMS). El hijo, a su vez, apunta al siguiente registro hijo virtual que tiene el mismo padre virtual (usando un *apuntador a gemelo lógico*). Esto es similar a los apuntadores al siguiente de un tipo de conjuntos en las bases de datos de red.
5. Registros ficticios de padre virtual. Como los VPHV no tienen nombres, es recomendable hacer que un registro sea padre virtual en *cuando más un* VPHV. Si el diseño lógico elegido tiene un tipo de registros que es un padre virtual en varios VPHV, se puede crear un tipo de registros ficticio que sea el hijo real de ese tipo de registros para cada VPHV adicional. Así, cada registro ficticio será padre virtual en *un solo* VPHV.

Las bases de datos jerárquicas también permiten dividir una jerarquía en grupos de tipos de registros por razones de eficiencia. Esto es similar al concepto de área de los SGBD de red tipo DBTG. Un grupo, denominado **grupo de conjunto de datos** en IMS, contiene un subárbol del esquema jerárquico y se hace corresponder con un área de almacenamiento físico contiguo. Esto mejora el acceso a los registros dentro del subárbol almacenándolos muy cercanos entre sí. Se puede proveer acceso directo a una raíz de un grupo así mediante un índice secundario.

14.4 Herramientas automatizadas de diseño*

Los diseñadores expertos todavía efectúan manualmente la mayor parte de las tareas implicadas en el diseño de bases de datos, aprovechando su experiencia y conocimientos en este proceso. Sin embargo, es difícil realizar a mano muchos aspectos del diseño de bases de datos, además de que éstos son susceptibles de automatización. Por ejemplo, es relativamente fácil automatizar una buena parte de la fase de transformación de modelos de datos. La detección de conflictos entre los esquemas antes de integrarlos puede dificultarse bastante si se hace a mano. De manera similar, la evaluación cuantitativa de diferentes alternativas para el diseño físico de la base de datos puede requerir mucho tiempo. Existen varias herramientas de diseño que ayudan con aspectos especiales del diseño de bases de datos, como los aspectos conceptuales, de transformación y físicos, así como con la recolección y análisis de requerimientos. No estudiaremos aquí las herramientas para el diseño de bases de datos; nos limitaremos a mencionar las siguientes características que debe poseer toda buena herramienta de diseño:

- Una interfaz fácil de usar: Esto es crucial porque permite a los diseñadores concentrarse en su tarea y no en entender la herramienta. Las interfaces gráficas y de lenguaje natural son las más difundidas. Las diferentes interfaces pueden adaptarse a los usuarios finales o a los diseñadores de bases de datos expertos.
- Componentes analíticos: Casi todas las herramientas cuentan con componentes analíticos para tareas que es difícil realizar manualmente, como evaluar alternativas de diseño físico o detectar restricciones contradictorias entre las vistas.
- Componentes heurísticos: Aspectos del diseño que no se pueden cuantificar con precisión pueden automatizarse adoptando reglas heurísticas en la herramienta de diseño. Estas reglas sirven para evaluar de manera heurística las alternativas de diseño.
- Análisis de ventajas y desventajas: Una herramienta debe presentar al diseñador un análisis comparativo adecuado siempre que ofrezca múltiples alternativas para elegir.
- Exhibición de resultados del diseño: Los resultados del diseño, como son los esquemas, a menudo se exhiben en forma diagramática. Otros tipos de resultados pueden mostrarse como tablas, listas o informes de fácil interpretación.
- Verificación del diseño: Ésta es una característica altamente deseable. Su propósito es verificar que el diseño resultante satisfaga los requerimientos iniciales.

Hoy día crece la convicción de que las herramientas de diseño son valiosas, y se están haciendo indispensables para resolver los problemas que se encaran al diseñar bases de datos grandes. El proceso de diseño ya no es concebible sin el apoyo de herramientas adecuadas para crear bases de datos de gran tamaño que abarquen a la organización en su conjunto. También está aumentando la convicción de que el diseño de esquemas y el de aplicaciones deben ir de la mano. Las herramientas CASE (ingeniería de software asistida por computador) que están apareciendo van dirigidas a ambas áreas. Algunas herramientas usan tecnología de sistemas expertos para guiar el proceso de diseño mediante la inclusión de conocimientos expertos en forma de reglas. Esta tecnología también es de utilidad en la fase de recolección y análisis de requerimientos, que suele ser un proceso laborioso y frustrante. La tendencia es hacia la utilización de diccionarios de datos y de herramientas de diseño para lograr mejores diseños de bases de datos complejas.

14.5 Resumen

En este capítulo examinamos las diferentes fases que han de cubrirse durante el diseño de bases de datos. También vimos el lugar que ocupan las bases de datos dentro de un sistema de información para la gestión de los recursos de información de una organización. El proceso de diseño de bases de datos abarca seis fases, pero las tres que suelen incluirse como la parte central del mismo son el diseño conceptual, el diseño lógico (transformación de modelos de datos) y el diseño físico. También analizamos la fase inicial de recolección y análisis de requerimientos, que muchas veces se considera una *fase de prediseño*. Además, en algún momento durante el diseño, es preciso elegir un paquete de SGBD específico. Estudiamos algunos de los criterios de la organización que influyen sobre la elección de un SGBD.

Destacamos la importancia de diseñar tanto el esquema como las aplicaciones (o transacciones). Examinamos varios enfoques del diseño de esquemas conceptuales y la diferencia entre el diseño de esquemas centralizado y el enfoque de integración de vistas. En la sección 14.3 analizamos los factores que afectan las decisiones del diseño físico de bases de datos y proporcionamos pautas para elegir entre las diversas alternativas de diseño físico para los SGBD relacionales, de red y jerárquicos. Por último, analizamos brevemente el empleo de herramientas de diseño automatizadas.

Preguntas de repaso

14.1. ¿Cuáles son las seis fases del diseño de bases de datos? Analice cada una de ellas.

14.2. ¿Cuáles de las seis fases se consideran las actividades principales del proceso de diseño de bases de datos propiamente dicho? ¿Por qué?

14.3. ¿Por qué es importante diseñar los esquemas y las aplicaciones en paralelo?

14.4. ¿Por qué es importante usar un modelo de datos independiente de la implementación durante el diseño de esquemas conceptuales?

14.5. Explique las características que debe poseer un modelo de datos para el diseño de esquemas conceptuales.

14.6. Compare y contraste los dos principales enfoques del diseño de esquemas conceptuales.

14.7. Analice las estrategias para diseñar un solo esquema conceptual a partir de sus requerimientos.

14.8. ¿Cuáles son los pasos del enfoque de integración de vistas para diseñar esquemas conceptuales? ¿Cómo funcionaría una herramienta de integración de vistas? Diseñe una arquitectura modular simple para una herramienta de este tipo.

14.9. ¿Cuáles son las diferentes estrategias para la integración de vistas?

14.10. Explique qué factores influyen sobre la elección de un paquete de SGBD para el sistema de información de una organización.

14.11. ¿Qué es la transformación de modelos de datos independiente del sistema?

14.12. ¿Cuáles son los factores importantes que influyen sobre el diseño físico de bases de datos?

14.13. Analice las decisiones que hay que tomar durante el diseño físico de una base de datos.

14.14. Explique qué son los ciclos de vida macro y micro de un sistema de información.

14.15. Analice las pautas para el diseño físico de bases de datos en los SGBD relacionales.

14.16. Analice las pautas para el diseño físico de bases de datos en los SGBD de red.

14.17. Analice las pautas para el diseño físico de bases de datos en los SGBD jerárquicos.

Bibliografía selecta

Se ha escrito muchísimo sobre el diseño de bases de datos. En primer término mencionaremos los libros que tratan el diseño de bases de datos. Wiederhold (1986) es un texto muy completo que cubre todas las fases del diseño de bases de datos, haciendo hincapié en el diseño físico. El proyecto DATAID de Italia, un proyecto muy amplio que aborda muchos aspectos del diseño de bases de datos, ha publicado dos libros (Ceri 1983; Albano *et al.* 1985). Batini *et al.* (1992), Teorey (1990) y McFadden y Hoffer (1988) destacan el diseño conceptual y lógico de bases de datos. Brodie *et al.* (1984) contiene varios capítulos sobre modelado conceptual, especificación y análisis de restricciones, y diseño de transacciones. Teorey y Fry (1982) presenta una metodología para el diseño de caminos de acceso en el nivel lógico. Yao (1985) es una colección de obras que abarcan desde las técnicas de especificación de requerimientos hasta la reestructuración de esquemas. El libro de Atre (1980) analiza la administración de bases de datos y los diccionarios de datos.

A continuación citamos referencias a artículos selectos que analizan los temas estudiados en este capítulo. Navathe y Kerschberg (1986) examinan todas las fases del diseño de bases de datos y destacan el papel de los diccionarios de datos. Goldfine y Konig (1988) y ANSI (1989) analizan el papel que tienen los diccionarios de datos en el diseño de bases de datos. Eick y Lockemann (1985) propone un modelo para la recolección de requerimientos. Rozen y Shasha (1991), Schkolnick (1978) y Carlis y March (1984) presentan modelos para el problema del diseño físico de bases de datos. March y Severance (1977) analiza la segmentación de registros. En Gane y Sarson (1979) y De Marco (1979) se estudian estrategias de diseño estructurado de aplicaciones. Whang *et al.* (1982) presenta una metodología para el diseño físico de SGBD de red.

Navathe y Gudgil (1982) definieron estrategias para la integración de vistas. Estas metodologías se comparan en Batini *et al.* (1986). Trabajos detallados sobre la integración *n*-aria de vistas pueden encontrarse en Navathe *et al.* (1989), Elmasri *et al.* (1986) y Larson *et al.* (1989). En Sheth *et al.* (1988) se describe una herramienta de integración basada en Elmasri *et al.* (1986). (1988). Otro sistema de integración de vistas se analiza en Hayne y Ram (1990). Casanova *et al.* (1991) describe una herramienta para el diseño modular de bases de datos. Motro (1987) examina la integración respecto a bases de datos ya existentes. La estrategia de integración de vistas binaria balanceada se analiza en Teorey y Fry (1982). Un enfoque formal para la integración de vistas, que utiliza dependencias de inclusión, se da en Casanova y Vidal (1982). Otros aspectos de la integración se estudian en Elmasri y Wiederhold (1979), Elmasri y Navathe (1984), Mannino y Effelsberg (1984) y Navathe *et al.* (1984*a*).

Todos los aspectos de la administración de bases de datos se examinan en Weldon (1981). En Curtice (1981) y Allen *et al.* (1982) se ofrecen panoramas sobre los diccionarios de datos. Algunas herramientas comerciales muy conocidas para diseñar bases de datos son ERMA, desarrollado por Arthur D. Little, Inc., en Cambridge, Massachusetts; la familia ADW de herramientas CASE, creadas por Knowledgeware en Atlanta, Georgia; MAST_ER de Infodyne, Inc.; y DDEW de CCA (Computer Corporation of America). Las herramientas de diseño automatizado se analizan en Bubenko *et al.* (1971), Albano *et al.* (1985), Navathe (1985), y en el capítulo 15 de Batini *et al.* (1992).

El diseño de transacciones es un tema que no se ha investigado de manera tan exhaustiva. Mylopoulos *et al.* (1980) propuso el lenguaje TAXIS, y Albano *et al.* (1987) crearon el sistema GALILEO, ambos sistemas muy completos para especificar transacciones. El lenguaje GORDAS para el modelo ECR (Elmasri *et al.* 1985) contiene un recurso para especificar transacciones. Navathe y Balaraman (1991) y Ngu (1991) estudian el modelado de transacciones en general para los modelos de datos semánticos.