Diego

Iberto Mendelzon es graduado de la Universidad de Buenos Aires y Master y PhD en la Universidad de Princeton. Hizo su post doctorado en el IBM T. J. Watson Research ter y, desde 1980 es profesor en la Universidad de Toronto.

sido científico visitante en numerosos laboratorios de investigación en ciencias de la iputación y ha participado como presidente o co-presidente en diversos programas y con-ncias internacionales relacionadas con el tema Base de Datos. También es Director de ıniación del ACM Special Interest Group on Management of Data y miembro del Comité ıtivo de SIGMOD.

ıublicado más de sesenta y cinco artículos en las áreas de diseño de bases de datos, ıajes de consulta, bases de conocimientos y sistemas de información sobre la WWW.

ılmente ha realizado consultorías para IBM Canadá, Bell-Northern Research, el Canadian ıerial Bank of Commerce, el Programa de las Naciones Unidas para el Desarrollo y ıue Canada Taxation, entre otras organizaciones.

uan Ale es graduado de la Universidad de Buenos Aires, donde obtuvo los títulos de Computador Científico, Licenciado en Ingeniería de Sistemas y Licenciado en Ciencias de ımputación. Es profesor en la Facultad de Ciencias Exactas y Naturales de dicha Univer-d desde 1977 y en otras universidades públicas y privadas de la Argentina. Actualmente ıofesor Titular de Bases de Datos en carreras de grado y posgrado.

1970 ha ocupado diversos cargos profesionales y de dirección, tanto en el área oficial privada, y desde 1990 se desempeña como consultor independiente, con actuación en ısas privadas y organismos gubernamentales.

participado en numerosos congresos nacionales e internacionales en calidad de expositor ıniembro de los comités organizadores.

1994 dirige un grupo de investigaciones en bases de datos que ha desarrollado tareas Universidad de Buenos Aires y, actualmente, en la Universidad de La Matanza, con ıs sobre bases de datos temporales, OLAP, data warehousing y data mining.

ın posee numerosas publicaciones científicas sobre estos temas.

Prentice Hall

Jesonica

Introducción a las bases de datos relacionales

MENDELZON - ALE -

Introducción a las bases de datos relacionales

Prentice Hall
Pearson Educación
Addison Wesley

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

Datos de catalogación bibliográfica

Nombre del autor: Mendelzon - Ale
Título: Introducción a las Bases de Datos Relacionales

PRENTICE HALL. Buenos Aires, 2000
ISBN: 987-97892-2-9
Materia: Sistemas

Formato: 17 x 23 Páginas: 288

29 MAR 2001

Gráfico S2 AL 034412

Gerente de División: Esteban Lo Presti
Editor: Darío Rubinstein
Arte de tapa y Diagramación de interiores: Pérez Villamil & Asociados

Revisión Técnica: Alejandro Vaisman

Este libro no puede ser reproducido total ni parcialmente en ninguna forma, ni por ningún medio o procedimiento, sea reprográfico fotocopia, microfilmación, mimeográfico o cualquier otro sistema mecánico, fotoquímico, electrónico, informático, magnético, electroóptico, etcétera. Cualquier reproducción sin el permiso previo por escrito de la editorial viola derechos reservados, es ilegal y constituye un delito.

© 2000, PEARSON EDUCATION S.A.
Av. Reg. de Patricios 1959, (1266) Buenos Aires, Rep. Argentina

ISBN: 987-97892-2-9

Queda hecho el depósito que dispone la ley 11.723
Impreso en la Argentina. Printed in Argentina

Primera edición: enero de 2000

CONTENIDOS

Prefacio xi

1. Funciones de los Sistemas de Bases de Datos 1
   1.1. Funciones de un SGBD 3
   1.2. El Administrador de Base de Datos 6
   1.3. Arquitectura de un SGBD 7
   Notas bibliográficas 8

2. Modelos de Datos 9
   2.1. Conceptos 10
   2.2. Modelo Entidad-Interrelación 13
   2.3. Modelo Relacional 18
   2.4. Representación Relacional de Entidades e Interrelaciones 23
   Ejercicios 24
   Notas bibliográficas 28

3. Álgebra y Cálculo Relacionales 29
   3.1. Álgebra Relacional 30
   3.2. Cálculo Relacional 40
   3.3. Equivalencia entre el Álgebra y el Cálculo 43
   3.4. Completitud Relacional 46
   Ejercicios 47
   Notas bibliográficas 50

El Lenguaje de Consulta SQL 51
4.1 Introducción 52
4.1.1. Definición de Datos 53
4.1.2. Consulta 54
4.2 Predicados de Comparación Cuantificados 57
4.3 La Operación de Junta en SQL 58
4.3.1. Consultas Anidadas 59
4.3.2. Otros Tipos de Junta 61
4.3.3 Outer Join 62
4.4. Otros Operadores 65
4.5. Una forma de Implementar el Operador de División 67
4.6. Uso de Funciones 69
4.7 Orden y Agrupamiento 71
4.8. Actualización 72
4.9. Definición de Vistas 74
4.10 Control de Seguridad 76
4.11 Valores Nulos 77
4.12 Restricciones 79
4.13 Programación SQL 86
4.14 SQL Dinámico 88
Ejercicios 90
Notas bibliográficas 98

Teoría del Diseño Relacional 99
5.1 Esquemas Indeseables 99
Redundancia 100
Anomalías de Actualización 100
Pérdida de Información 101
5.2 Dependencias Funcionales 103
Implicación de Dependencias Funcionales 104
Reglas de Inferencia Adicionales 106
Clausura de un Conjunto de Atributos 107
Equivalencia de Conjuntos de dfs 109
Cubrimiento Mínimal para un Conjunto de Dependencias Funcionales 109
5.3 Propiedades Deseables de los Esquemas 112
Descomposición sin Pérdida 112
Preservación de Dependencias 117
Forma Normal de Boyce-Codd 121
Tercera Forma Normal 124
Otras Formas Normales 126

5.4. Dependencias Multivaluadas 127
Axiomas y Reglas de Inferencia para Dependencias Multivaluadas 129
Base de Dependencias 131
Cuarta Forma Normal (4FN) 132
Dependencias Multivaluadas Embebidas 135
5.5. Dependencias de Junta 136
Quinta Forma Normal (5FN) 137
Implicación de Dependencias 137
Ejercicios 138
Notas bibliográficas 143

6. Estructuras Físicas de Datos 145
6.1. Archivos, Registros, Bloques 145
6.2. Manejo de Buffers 147
6.3 Representación de Relaciones 148
Diccionario de Datos 150
6.4. Indexación 151
Método Secuencial Indexado 152
Índices Densos 153
Árboles-B 153
6.5 Hashing 159
6.6 Hashing Extensible 160
6.7. Comparación de los Métodos 168
6.8. Accesos por Campos No-Clave 169
Ejercicios 170
Notas bibliográficas 174

7. Procesamiento de Consultas 175
7.1. Introducción 175
Costo de Procesamiento 176
Etapas en el Procesamiento de Consultas 177
7.2. Transformación de Expresiones Algebraicas 178
7.3. Estimaciones del Costo de Procesamiento 180
Uso de Índices 182
7.4 Métodos de Cálculo de Juntas 185
Iteración Ingenua 185
Iteración por Bloques 186
Método *Sort-Merge* 187

Uso de Índices 187
Método *Hash* 188
7.5. Junta de Múltiples Relaciones 189
7.6. Método de Índice de Junta con Mapas de Bits 191
Ejercicios 192
Notas bibliográficas 196

8. Control del Paralelismo y Recuperación 197
8.1. Introducción 197
8.2. Propiedades ACID 200
8.3. Ejecución de Transacciones 201
8.4. Candados y Exclusión Mutua 204
8.5. Caracterización de Ejecuciones Serializables 206
8.6. Cierre de Dos Fases 210
8.7. Niveles de Aislamiento 212
8.8. Recuperación Transaccional 214
8.9. Gestión de *Logs* o Bitácoras 216
8.10. El Gestor de Recuperación 219
8.11. Algoritmos de Recuperación 222
8.12. *Checkpoints* o Puntos de Control 224
8.13. Recuperación de Medio 224
Ejercicios 226
Notas bibliográficas 230

Apéndice: El Estándar SQL:1999 231
A.1. Introducción 231
A.2. Características Relacionales 232
A.2.1. Nuevos Tipos de Datos 232
A.2.2. Nuevos Predicados 236
A.2.3. Nueva Semántica 236
A.2.4. Seguridad 238
A.2.5. Base de Datos Activa 240
A.3 Características Relacionadas con Orientación a Objetos 242
A.3.1. Tipos Estructurados Definidos por el Usuario 242
A.3.2. Módulos Persistentes Almacenados 246
A.4. Implementaciones 250
Notas Bibliográficas 250

Bibliografía 251

Índice 267

relación s.f.pl. *Arg., Bol., Urug.* En ciertos bailes folclóricos, versos que se dicen uno a otro los integrantes de una pareja.

PREFACIO

En 1976, el ilustre E.W. Dijkstra escribía, después de haber asistido a un Congreso sobre bases de datos:*

> *Aprendí algunas cosas sobre bases de datos. Aprendí –o mejor dicho, confirmé mi impresión preliminar– que el término "Tecnología de Base de Datos", aunque a veces se usa, es inmaduro, pues casi no existe una "ciencia" detrás que justifique el uso del término "tecnología". Hasta tengo mis dudas cuando se me pide que crea que "la tecnología de bases de datos aún está en su infancia", porque esto me suena como pedirme que considere a los charlatanes de feria la infancia de la ciencia médica.*

En este libro veremos que en estos veintitantos años tal ciencia ha empezado a desarrollarse. En buena medida tal desarrollo se debió a la introducción del modelo relacional por E.F. Codd en 1970; por esta razón, y por el papel dominante que juegan hoy en día las bases de datos relacionales, hemos decidido restringir el libro a éstas. Sin embargo, los conceptos y las técnicas que contiene son aplicables y han sido aplicadas a otros modelos de datos también.

Por razones de espacio y de tiempo, este volumen dista de ser un tratamiento exhaustivo del área; hemos seleccionado las ideas que nos parecen fundamentales y esperamos que el lector interesado siga las sugerencias de la bibliografía que encontrará al final de cada capítulo. En particular los te

* E.W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag Berlin, 1982, p. 228, nuestra traducción.

nas de diseño físico de las bases de datos que apenas trataremos y el de ba-
es de datos paralelas y distribuidas, que no trataremos en absoluto mere-
cen estudio detenido.

El propósito de este libro es el de servir de texto en los cursos introducto-
rios de bases de datos de las carreras de Ciencias de la Computación e Inge-
niería Informática.

Se ha tratado de mantener el tratamiento de los distintos temas en un ni-
vel accesible con un estilo simple y claro. Al final de cada capítulo se encuen-
tran ejercicios que refuerzan los conceptos introducidos. Los ejercicios son
de dificultad variada ya que comienzan con casos sencillos y llegan a un ni-
vel interesante de complejidad.

El contenido del libro se ha organizado en ocho capítulos y un apéndice. El
Capítulo 1 contiene la motivación para las bases de datos, una introducción
general al tema y la presentación de los restantes capítulos. El Capítulo 2 in-
troduce los conceptos sobre modelos de datos en general y presenta el mode-
lo entidad-interrelación, el modelo relacional y la representación relacional de
entidades e interrelaciones. Los lenguajes teóricos del modelo relacional son
presentados en el Capítulo 3, comenzando con el álgebra, luego el cálculo re-
lacional y la equivalencia entre ambos. SQL es tratado en el Capítulo 4, cu-
briendo las distintas funciones del lenguaje, sus capacidades y su sintaxis con
numerosos ejemplos ajustado al estándar ANSI SQL-92. El Capítulo 5 está de-
dicado a la Teoría del Diseño Relacional y en él se ha intentado proporcionar
un claro, preciso y bien motivado fundamento para la normalización de bases
de datos, con algoritmos suficientemente detallados para resolver casi todos
los problemas planteados en la teoría. Estos primeros cinco capítulos se refie-
ren básicamente al diseño y explotación de las bases de datos relacionales. A
partir del capítulo siguiente el foco se ha desplazado a aspectos propios e in-
ternos de un Sistema de Gestión de Bases de Datos (sgbd). En el Capítulo 6 da-
mos una breve síntesis de los métodos de organización de archivos que suelen
pueden utilizarse en la implementación de un sgbd relacional. El procesa-
miento de consultas por parte de un sgbd es analizado en el Capítulo 7, donde
tratamos la transformación de expresiones algebraicas, estimaciones de cos-
tos, estrategias de procesamiento y diversos métodos de cálculo de juntas. El
capítulo 8 está dedicado al procesamiento transaccional, por lo que se anali-
zan las propiedades ACID de las transacciones, se utilizan conceptos de teoría
de la serializabilidad para analizar su procesamiento en paralelo o concurren-
te, se introduce el protocolo de cierre de dos fases, los niveles de aislamiento
planteados en el SQL-92, como así también elementos de técnicas de recupe-
ración en particular el método *Deshacer/Rehacer*. El apéndice describe some-

ramente el futuro nuevo estándar para SQL, el SQL:1999, antes conocido co-
mo SQL3.

El material del libro, que incluye una intensiva ejercitación, se adapta pa-
ra su utilización en cursos de duración anual o cuatrimestral, con cargas ho-
rarias entre cien y ciento cincuenta horas totales.

La primera versión de este libro surgió de un curso dictado en la Prime-
ra Escuela Argentino-Brasileña de Informática, realizada en la Universidad
de Campinas en febrero de 1986. Esta versión fue extensivamente actualiza-
da y expandida entre 1998 y 1999, duplicando el tamaño original. A. Mendel-
zon agradece a los organizadores de dicha Escuela por haber hecho posible
la génesis del libro, y al *Computer Systems Research Group* de la Universidad
de Toronto y el *Istituto di Analisi dei Sistemi ed Informatica del Consiglio Na-
zionale delle Ricerche* de Italia por haber prestado sus facilidades para la pre-
paración del texto original. También el agradecimiento a Laura Lozano de
Celis por su asistencia en el dictado del curso y a Malisa Otermin por haber
leído y corregido el manuscrito. Mauricio Minuto Espil y Alejandro Vaisman
han contribuido con la preparación de muchos de los ejercicios que se en-
cuentran al final de cada capítulo. Agradecemos también a Jorge Eterovic,
de la Biblioteca del Maestro en Buenos Aires, por su valiosa asistencia en la
digitalización del manuscrito original y a Nora Gigante por su colaboración
en la construcción del índice alfabético. Juan Ale agradece a su esposa Ma-
ría Rosa y a sus hijos Gabriela, Verónica y Javier por su paciencia y compren-
sión en las horas sustraídas a su compañía.

Toronto y Buenos Aires,
septiembre de 1999.

CAPÍTULO 1

FUNCIONES DE LOS SISTEMAS DE BASES DE DATOS

El concepto de base de datos surgió a fines de la década del 60 como propuesta de solución a un conjunto de problemas técnicos y administrativos presentados en el manejo de archivos. A medida que los sistemas de información se volvían más complejos y se extendían a nuevas áreas de la operación de una empresa o institución, las dificultades en mantener su funcionamiento y costo bajo control se acentuaban. Por ejemplo, éstos son algunos de los problemas surgidos:

- Redundancia: Si una organización decide registrar y almacenar cierto tipo de información para una aplicación específica, es muy probable que tarde o temprano esa información sea requerida no sólo para la aplicación que se tenía en mente originalmente, sino para otras también. Por ejemplo, un banco puede necesitar datos personales de sus clientes a fin de poder enviarles resúmenes de cuenta generados automáticamente; más adelante, tales datos pueden también resultar útiles para decidir a qué clientes enviar material promocional, o para determinar el tratamiento impositivo de intereses cobrados, o para ubicar clientes cuyo domicilio es cercano a cierta sucursal. Si cada una de estas aplicaciones se trata como un proyecto separado, y no hay coordinación que permita descubrir los requerimientos de información comunes a varias aplicaciones, es probable que la misma información se duplique en varios archivos. Esto tiene dos consecuencias no deseables. En primer lugar, la redundancia resultará en costos de almacenamiento mayores que los necesarios. En segundo lugar, y más importante, la redundancia introduce la posibilidad de *incon-*

sistencia: si un cliente cambia de dirección, ¿cómo garantizamos que todos los sistemas que usan información sobre direcciones de clientes se enteran de esta modificación?

- Dificultad de Acceso: El diseño de un sistema de aplicación generalmente parte de la base de que la información será utilizada de ciertas formas y no de otras. Por ejemplo, el sistema que genera resúmenes de cuenta puede necesitar acceso eficiente al nombre y dirección del titular dado el número de cuenta, pero probablemente no proveerá la función de determinar qué clientes viven en el área definida por cierto código postal. Cada pedido de información de parte de los usuarios implica la intervención de personal especializado en sistemas para definir e implementar los programas que generan esa información.
- Dependencia de la Estructura de Almacenamiento: Como mencionamos en el punto anterior, un sistema de aplicación hará ciertas suposiciones basadas en los requerimientos de la aplicación; pero además, hará suposiciones basadas en la organización física de los archivos que utiliza. Por ejemplo, la lógica de búsqueda del registro de un cliente dado su número de cuenta será muy distinta según si el archivo esté almacenado secuencialmente en orden de número de cuenta, o accesible mediante hashing sobre el número de cuenta, o indexado secuencial. A más bajo nivel, los analistas y programadores pueden incorporar en los sistemas que construyen suposiciones sobre las características físicas de los periféricos usados para los archivos, tales como el tamaño de una pista o bloque de disco. La dificultad que ésta introduce es que los sistemas se vuelven altamente sensibles a los cambios en su entorno: si cambiamos el método de organización de un archivo, o reemplazamos la unidad de disco por un nuevo modelo, necesitamos modificar, a veces en forma drástica, la lógica de todos los programas de aplicación que usan los archivos afectados por estos cambios. Este estado de cosas es lo que se llama *dependencia de datos*, y su ausencia, que es uno de los objetivos básicos del uso de sistemas de base de datos, se llama *independencia de datos*.

Para tratar de evitar este tipo de problemas, la tecnología de bases de datos hace la siguiente propuesta. Tomemos el conjunto de toda la información que es relevante a la operación de una organización: para un banco, será información relativa a cuentas, clientes, empleados, equipos, etc. Toda esta información, apropiadamente organizada y codificada, se colocará en un gran reservorio llamado base de datos. Ningún programa de aplicación tendrá acceso directo a los archivos que componen la base de

datos, sino que interpondremos entre estos archivos y los programas de aplicación un nuevo nivel de software llamado *sistema de gestión de bases datos*, que abreviaremos *sgbd*. El sgbd provee acceso a la información a un alto nivel de abstracción: en lugar de manipular archivos, registros, índices, cilindros, el programa de aplicación se maneja enteramente en términos de clientes, cuentas, saldos, etc., que son traducidos por el sgbd a su implementación física. La base de datos no estará bajo el control de ninguno de los grupos encargados de construir sistemas de aplicación, sino que tal control estará centralizado en un grupo o persona, el administrador de base de datos, abreviado abd, quien es responsable de definir, mantener, y garantizar la integridad y seguridad de la base de datos. Finalmente, el sgbd proveerá generalmente una aplicación especial llamada lenguaje de consulta que permite a los usuarios finales interrogar la base de datos a discreción sin tener que pasar por el analista o programador como intermediarios.

No es difícil ver que este esquema, al menos en teoría, elimina los problemas mencionados más arriba. La redundancia se elimina, o se reduce al nivel deseado, gracias al control centralizado de qué información será almacenada y en qué forma. El acceso a la información por parte de los usuarios finales se hace más rápido y flexible por el uso del lenguaje de consulta. El nivel de abstracción provisto por el sistema hace que los programas de aplicación sean relativamente independientes de los detalles físicos de almacenamiento, ya que tales dependencias están localizadas en módulos del sgbd.

Debemos notar que esta discusión simplifica un proceso que es en la práctica complejo. El ideal de la base de datos institucional, única y omnisciente, está lejos de haberse convertido en realidad, e incluso es a veces indeseable la centralización excesiva. Pero en líneas generales, hemos dado la motivación para la introducción de un sgbd en una organización. El resto del libro estará dedicado a describir en detalle los aspectos técnicos de tales sistemas y de su uso efectivo. En lo que queda de este capítulo, discutiremos en general las funciones y arquitectura de estos sistemas.

1.1. FUNCIONES DE UN SGBD

Las funciones provistas por un sgbd se pueden agrupar en tres clases:

- Consulta y Actualización de los Datos
- Mantenimiento de Esquemas
- Manejo de Transacciones

Consulta y Actualización

Ésta es la clase más básica de funciones y la única que es visible "desde afuera". Consiste en un conjunto de herramientas que permite a los distintos tipos de usuarios del sgbd extraer, manipular y modificar la información almacenada en la base de datos.

Conviene clarificar aquí qué queremos decir cuando hablamos de *usuarios* de un sgbd. Podemos distinguir por lo menos tres clases de usuarios.

El *usuario casual* o usuario final es una persona sin conocimientos técnicos sobre bases de datos ni sobre programación y computación en general. Los objetivos de interfaces para este tipo de usuario son permitirle obtener la información que busca con un mínimo de entrenamiento previo y en forma indolora. Típicamente la interacción entre usuarios finales y el sgbd se dará a través de una terminal, en forma interactiva.

El programador de aplicación es el encargado de construir sistemas que utilizan el sgbd. Generalmente su interés no es interactuar directamente con el sgbd, sino escribir programas que invocan las funciones necesarias para acceder a la base de datos.

Finalmente, el administrador de la base de datos es el usuario todopoderoso, que necesita íntima familiaridad con los mecanismos internos del sistema, uso de recursos, performance, requerimientos de seguridad, etc.

No incluimos dentro de nuestra noción de usuarios a aquellos que solamente interactúan con el sistema en formas estereotipadas, mediadas por un programa de aplicación, como por ejemplo un cajero de banco o un agente en un mostrador de aerolínea. La tarea de los programadores de aplicación es empaquetar el conjunto de funciones del sgbd que este tipo de usuario requiere, para que pueda invocarlas simplemente oprimiendo una tecla.

Algunos sistemas proveen una única interfaz, llamada generalmente lenguaje de consulta, para ser usada por todos los tipos de usuarios descriptos arriba. Por ejemplo, el lenguaje SQL permite expresar consultas y actualizaciones en una notación vagamente similar al lenguaje natural. La expresión

```
select numero, nombre
from alumno
where carrera = "MAT" and edad < 21
```


produce como resultado una tabla con los números y nombres de todos los alumnos menores de edad en la Carrera de Matemática. El mismo lenguaje puede ser usado desde un programa escrito digamos en COBOL, y también el mismo lenguaje provee funciones de administración para ser usadas por el abd. En el Capítulo 4 describiremos este lenguaje en más detalle.

Otros sistemas en cambio proveen interfaces de estilos muy distintos para los distintos tipos de usuario, reconociendo especialmente que los usuarios finales necesitan interfaces visuales radicalmente distintas de las que puede requerir un programador.

Mantenimiento de Esquemas

El *esquema* de la base de datos es la descripción de la estructura de la información almacenada en ella. Por ejemplo, para un sistema basado en tablas, el esquema puede consistir en una lista de las tablas en uso, los campos que contienen, el tipo de datos de cada campo, descripciones en lenguaje natural del propósito de cada tabla y cada campo, y restricciones sobre los valores admisibles en cada campo.

Así como los usuarios necesitan acceder, agregar y modificar datos, también necesitan acceder, agregar y modificar el esquema de datos. Por ejemplo, un usuario que se acerca por primera vez a una base de datos querrá saber antes que nada qué información contiene ésta, un programador puede escribir programas que definan y creen nuevos tipos de entidades, o eliminen algunos preexistentes; el abd necesita controlar qué usuarios tienen acceso a qué información, formulando reglas de seguridad que se hacen parte del esquema.

Manejo de Transacciones

Una de las áreas principales de aplicación de los sgbd's es lo que se llama procesamiento de transacciones. Una transacción es un programa de aplicación, generalmente de duración breve, que accede y actualiza una parte también generalmente pequeña de la base de datos. Típicos ejemplos son un depósito o extracción de una cuenta bancaria, o una reservación en un vuelo, o una verificación de una tarjeta de crédito. El manejo de transacciones consiste en controlar múltiples transacciones ejecutando en *paralelo* sobre una misma base de datos corriendo en un sistema que puede *fallar*. Los objetivos

del gestor de transacciones del sgbd son: evitar que las transacciones interfieran unas con otras al ejecutar en paralelo, y garantizar que la base de datos no sea dañada en forma irreparable por caídas, ya sea del sistema en sí o de algunas de las transacciones. El primero de los objetivos da lugar a lo que se llama *control del paralelismo*; el segundo, a *técnicas de recuperación*.

1.2. El Administrador de Base de Datos

El abd es la persona o grupo que centraliza el control de la base de datos y de los programas de aplicación que la acceden. Las funciones del abd incluyen:

- **Definición del Esquema:** Antes de poder almacenar información, es necesario definir la estructura lógica de ésta. Esta tarea se puede asemejar a escribir las declaraciones para un programa complejo en un lenguaje de alto nivel, con la diferencia de que el abd no sabe exactamente qué función va a cumplir el programa y debe seleccionar un conjunto de estructuras de datos que satisfagan lo mejor posible los requerimientos presentes y futuros. Esta tarea se conoce como *diseño de bases de datos*; en el Capítulo 5 discutiremos algunas ideas teóricas destinadas a ayudar en el proceso de diseño.
- **Definición de Estructuras de Almacenamiento:** A fin de obtener el desempeño adecuado, el abd debe seleccionar del repertorio de estructuras físicas provisto por el sgbd las que parezcan más adecuadas. El Capítulo 7 presenta las estructuras de almacenamiento más comunes.
- **Modificación de Estructuras Físicas y Lógicas:** A medida que los requerimientos y el entorno cambian, el abd necesitará modificar las decisiones mencionadas en los dos puntos anteriores.
- **Autorización de Acceso:** El abd es responsable por determinar qué usuarios o clases de usuarios tendrán acceso a qué información y qué clase de acceso: algunos usuarios podrán leer pero no actualizar cierta tabla, otros tendrán acceso irrestricto, otros ni siquiera sabrán que la tabla existe. En el Capítulo 4 describiremos las facilidades que provee para esto el lenguaje SQL.
- **Especificación de Restricciones:** El sistema mantiene un conjunto de reglas que restringen los valores que pueden aparecer legalmente en la base de datos, tales como "el valor de *edad* es un entero entre 16 y 99" o "cada número de pasaporte pertenece a una única persona". Estas reglas se almacenan en estructuras de datos especiales que el sistema consulta ante cada actualización. La especificación de estas reglas es responsabilidad del abd.

1.3. Arquitectura de un SGBD

La arquitectura de los sistemas de bases de datos no se ha estandarizado al mismo grado que la de un compilador, pero hay ciertos módulos que de un modo u otro forman parte de la mayoría de los sistemas, incluyendo:

- **Gestor de Archivos:** Éste es el módulo que interactúa directamente con el sistema de archivos provisto por el sistema operativo. Tiene a su cargo el manejo de las estructuras físicas de almacenamiento, administración de espacio en memoria secundaria, y métodos eficientes de acceso y búsqueda. En algunos casos los diseñadores del sgbd, estimando que el manejo de archivos provisto por el sistema operativo no es adecuado a los fines del sgbd, eligen implementar sus propias rutinas de acceso a archivos directamente sobre el hardware.
- **Gestor de Datos:** Ubicado a un nivel de abstracción mayor que el gestor de archivos, este módulo provee las funciones fundamentales que sirven de interfaz entre los datos y los programas de aplicación o consultas de los usuarios. Por ejemplo, funciones típicas provistas a este nivel serían las de encontrar todos los registros, o verificar que un conjunto de datos satisfaga las restricciones semánticas del diseño.
- **Procesador de Consultas:** Este módulo traduce las consultas expresadas en un lenguaje de consulta de alto nivel a una sucesión de pedidos básicos al nivel que el gestor de datos interpreta. Esta traducción no es trivial porque, en general, existen muchas estrategias distintas para implementar una consulta de alto nivel, con costos muy diferentes. El procesador de consultas tiene a su cargo la *optimización de consultas*, o sea, trata de hallar la implementación más eficiente de una consulta dada. En el Capítulo 7 presentaremos las ideas básicas del procesamiento de consultas.
- **Controlador del Paralelismo:** Cuando el sgbd soporta múltiples procesos accediendo y posiblemente modificando una misma base de datos en paralelo, es importante garantizar que estos procesos no interfieran unos con otros, manteniendo a la vez el mayor grado de paralelismo posible. Ésta es la función del controlador del paralelismo, que discutiremos en más detalle en el Capítulo 8.

- Sistema de Recuperación: Junto con el controlador del paralelismo, este módulo implementa la función que llamamos de manejo de transacciones. El sistema de recuperación debe mantener suficiente información acerca del progreso de las distintas transacciones en ejecución para poder garantizar que la base de datos será restaurada a un estado consistente en caso de una falla parcial o total. Este tema también será tratado en más detalle en el Capítulo 8.

NOTAS BIBLIOGRÁFICAS

Existen varios excelentes libros de texto sobre sistemas de bases de datos en general; entre ellos es ya un clásico el de J. D. Ullman [202] editado en dos volúmenes. Otros son A. Silberschatz, H. F. Korth y S. Sudarshan [191]; R. Ramakrishnan y J. Gehrke [174]; P. O'Neil [167]; R. Elmasri y S. Navathe [80] y también en dos volúmenes la obra de C. Date [71] que va por la sexta edición.

CAPÍTULO 2

MODELOS DE DATOS

Como ya hemos dicho, una de las ventajas clave en el uso de un sgbd es nivel de abstracción que éste provee en el acceso a la información, d mismo modo que un lenguaje de programación como PASCAL permite al pr gramador expresar sus algoritmos en términos de registros, arreglos, listas árboles, en lugar de bits y direcciones de memoria. Cada sgbd provee a s usuarios un vocabulario básico de conceptos en términos de los cuales ésto pueden definir y manipular la información a ser almacenada. Este vocab lario se denomina el modelo de datos. Volviendo a los lenguajes de prog mación, podríamos decir que el modelo de datos ofrecido por FORTRAN co siste esencialmente en números enteros, reales y complejos, vectores matrices de éstos, y operaciones aritméticas elementales sobre ellos. En cam bio el modelo de datos de un lenguaje como LISP consiste en estructuras lista y operaciones de extracción y construcción de las mismas.

¿Qué tipo de modelo de datos se utiliza en los sgbd's? Existe una varied considerable en los sistemas disponibles comercialmente, y una enorme p liferación de propuestas en la literatura de investigación: existen mode que presentan los datos en términos de tablas, grafos, redes, árboles, func nes, conjuntos, etc. En la próxima sección trataremos los conceptos relac nados con los modelos de datos en general. En la siguiente sección pres taremos el modelo de Entidad-Interrelación (Entity-Relationship) o E propuesto por Peter Chen en 1976. Vamos a describirlo en términos relati mente informales, sin tratar de dar definiciones precisas. Cuando traten el modelo relacional veremos cómo es posible dar definiciones matemáti precisas de los conceptos de un modelo de datos.

2.1. CONCEPTOS

Un modelo, en general, es una representación simplificada de un sistema real. Si la representación es adecuada, podremos interrogar al modelo si deseamos conocer alguna propiedad del sistema real. Luego una base de datos es un modelo del conjunto de los datos relativos a una organización o al menos a una aplicación en esa organización.

Un modelo de datos debe poder representar tanto las características estáticas como las dinámicas del sistema real que se pretende modelizar. En consecuencia, un modelo de datos se define por los siguientes componentes:

i) Un conjunto de objetos y sus interrelaciones. Esto representa las características estáticas o invariantes e incluye las propiedades de los objetos.

ii) Un conjunto de operaciones o lenguaje, que representa las características dinámicas.

iii) Restricciones sobre los objetos, sus interrelaciones y las operaciones definidas sobre ellos.

El conjunto de objetos y sus interrelaciones resultan de abstracciones realizadas sobre la estructura del sistema real y constituye de este modo la estructura del modelo. Ejemplos de objetos en los modelos de datos son entidades, tablas, registros, eventos, estados, etc.

Es posible agrupar los objetos de la misma clase en tipos. De esta manera, los objetos individuales son instancias de un tipo. También es posible agrupar diferentes tipos para obtener otro tipo, en un nivel de abstracción superior. Un tipo se define por una caracterización de propiedades estructurales y comportamentales compartidas por una colección de objetos. Por otra parte, una instancia de un tipo es un objeto particular que tiene las propiedades características del tipo. Por ejemplo, los distintos automóviles forman el tipo AUTOMÓVIL, los tipos AUTOMÓVIL, ÓMNIBUS, TREN, AVIÓN, etc. forman el tipo VEHÍCULO.

Un objeto está formado por objetos de menor nivel, los que a su vez están interrelacionados entre sí de alguna forma. En nuestro proceso de modelización es necesario identificar los objetos primitivos o primarios, los que son considerados irreducibles o atómicos, con lo que desestimamos la descomposición de dichos objetos en subobjetos.

Lo anterior se expresa por medio de los siguientes principios de abstracción, los que permiten estructurar el conjunto de objetos en un modelo de datos:

i) Generalización: permite agrupar tipos similares en un tipo genérico de nivel superior. El proceso de abstracción consiste en desestimar las diferencias mientras que se enfatizan las similitudes. Por ejemplo, el tipo VEHÍCULO es una generalización de los tipos AUTOMÓVIL Y BUQUE. Da lugar a la relación "es_un" entre un tipo particular y su supertipo.
Especialización: es el proceso opuesto al de generalización y permite descomponer un tipo en subtipos. Por ejemplo AERONAVE es un tipo genérico y los subtipos son AVIÓN, GLOBO, etc.

ii) Clasificación: es el proceso por el cual se agrupa en un tipo a una colección de objetos que comparten el mismo conjunto de propiedades relevantes. Por ejemplo, un conjunto de autos forman el tipo AUTOMÓVIL. Da lugar a la relación "instancia_de" entre los objetos y su tipo.
Instanciación: constituye el recíproco de clasificación y permite considerar los objetos integrantes de un tipo en forma particular. Así, por ejemplo, del tipo NAVES de una compañía podemos considerar un avión particular.

iii) Agregación: es un proceso de abstracción por el cual se construye un agregado u objeto de orden superior, a partir de la consideración de las interrelaciones entre sus objetos constitutivos. En general, estos objetos son heterogéneos. Por ejemplo, un automóvil es un agregado que se integra a partir del chasis y la carrocería. Esto establece la relación "parte_de" entre los objetos parte y el agregado.
Desagregación: es el proceso inverso al de agregación.

iv) Asociación: en esta forma de abstracción, una interrelación entre objetos similares homogéneos se considera como un objeto conjunto de nivel superior. Por ejemplo, un conjunto de jugadores forman un equipo. Establece una relación "miembro_de" entre un objeto y su objeto conjunto.
Disociación: es el proceso inverso al de asociación. Permite considerar los miembros de un objeto conjunto en forma individual.

Las interrelaciones entre objetos, por otra parte, se analizan en términos de sus tipos. Así, si se tiene en cuenta el número de tipos de objetos que participan tenemos relaciones *unarias*, *binarias*, *ternarias* y, en general, *n-arias* si partici-

pan n tipos. Por otra parte, si se considera con cuántos objetos de un tipo se relaciona un objeto de un tipo dado, las interrelaciones se clasifican en tres tipos básicos: uno a uno (1:1), muchos a uno (M:1) y muchos a muchos (M:N).

El *conjunto de operaciones* puede dividirse primariamente en dos grupos:

- Operaciones de manipulación que permiten crear, modificar y destruir tanto los objetos como sus interrelaciones, y
- Operaciones de consulta que permiten identificar objetos por medio de sus propiedades.

El primer tipo de operación modifica el estado de la base de datos al alterar los objetos o sus propiedades, mientras que el segundo puede alterar solamente los mecanismos de control que posibilitan la búsqueda.

Un conjunto perfectamente definido de operaciones se denomina una *transacción*. En general, una transacción se compone de acciones elementales sobre objetos individuales. Este concepto es de fundamental importancia para asegurar la consistencia entre operaciones atómicas concurrentes y requiere que todos los accesos a la base de datos estén dentro de transacciones.

Existen casos en que no resulta posible representar en un modelo todas las propiedades deseadas o requeridas por medio de los objetos, con sus atributos e interrelaciones, y las operaciones sobre esos objetos. En consecuencia, resulta necesario incorporar restricciones adicionales que expresen dichas propiedades.

Las propiedades mencionadas se refieren, en general, a aspectos semánticos y de integridad. En el primer caso las restricciones permiten representar en forma más precisa situaciones de la realidad. En el segundo, permiten restringir el conjunto de posibles estados de una base de datos. Las restricciones han sido clasificadas en los siguientes tres tipos básicos:

- Restricciones inherentes: constituyen las propiedades semánticas básicas del modelo de datos. Son aspectos que se representan directamente en la estructura del modelo. Un ejemplo es la representación de la estructura jerárquica de una organización mediante una estructura de árbol en el modelo.
- Restricciones explícitas: son las propiedades que deben ser expresadas necesariamente mediante un mecanismo destinado al efecto. Por ejemplo, utilizando un lenguaje del cálculo de predicados se puede expresar que un empleado no puede ganar más que su gerente o que un empleado trabaja para un único departamento.

- Restricciones implícitas: son las que se derivan de otras restricciones, ya sea inherentes o explícitas. Siguiendo con el ejemplo anterior y si además se explícita que un departamento tiene un único gerente, una restricción implícita es la que expresa que un empleado tiene un único gerente.

Los diferentes modelos de datos se distinguen básicamente por sus restricciones inherentes y explícitas. Ciertos tipos de asociaciones entre objetos de un modelo pueden ser forzadas por las clases de estructuras usadas por el modelo, mientras que en otros casos deben ser declaradas explícitamente. Un ejemplo es el caso de las relaciones funcionales.

Las restricciones explícitas pueden ser clasificadas a su vez como restricciones estáticas y restricciones dinámicas.

Las restricciones son aplicables a los objetos, a sus interrelaciones, a las interrelaciones entre sus atributos, como así también a las operaciones posibles.

Las restricciones estáticas especifican, en general, estados válidos de una base de datos (bd). Por ejemplo, no es posible asignar un sueldo a un empleado por debajo de un mínimo prefijado. Un ejemplo de restricción aplicado a objetos es la llamada *Integridad de Entidad*, por la cual un objeto no puede estar duplicado en la bd. Para el caso de las interrelaciones, el ejemplo es la llamada *Integridad Referencial*, por la cual si un objeto es referenciado por otro objeto, este último sólo puede existir si existe el primero. Como ejemplo de restricción a la interrelación entre atributos tomemos el caso de una persona con atributos Documento y Nombre. La restricción denominada *dependencia funcional* establece que si en la bd existen dos personas con el mismo documento, entonces deben ser la misma persona, por lo que los nombres serán necesariamente coincidentes.

Las restricciones dinámicas limitan las posibles transiciones entre estados de la bd. Una forma de implementar este tipo de restricción es mediante un mecanismo de verificación de condiciones previas y posteriores. Un ejemplo de restricción aplicado a una transición es la que impide reducir el sueldo de un empleado.

2.2. MODELO ENTIDAD-INTERRELACIÓN

Los conceptos de este modelo son de alto nivel semántico, cercanos al universo de discurso del usuario y no al del sistema. Chen lo propuso originalmente, no como un modelo de datos a ser implementado en la práctica por un sgbd, sino como una herramienta conceptual útil en el diseño de una base de datos,

pero tal que una descripción en términos de entidad-interrelación debería ser traducida a uno de los modelos de más bajo nivel antes de ser implementada en una computadora. Diversas herramientas de software conocidas como Herramientas CASE (Computer-Aided Software Engineering) realizan la traducción en forma automática al modelo relacional, generando las sentencias de creación de la base de datos con sus tablas y restricciones de integridad en SQL.

Entidades e Interrelaciones

Un modelo de entidad-interrelación es una descripción del aspecto del mundo real que el diseñador intenta capturar en un sistema de información. En este modelo, la estructura se describe por un conjunto de entidades u objetos de interés, relacionados entre sí por interrelaciones. Usaremos el término entidad en sentido genérico, refiriéndonos a una clase de objetos en lugar de a un objeto en particular. Por ejemplo, las siguientes entidades podrían ser de interés en alguna u otra aplicación:

clientes
cuentas corrientes
alumnos
tangos
intérpretes

Una instancia de una entidad es un individuo particular que pertenece a la clase definida por esa entidad; por ejemplo:

el cliente Hugo Sánchez
la cuenta corriente número 06-3152
la alumna Beatriz Guido
el tango "Mi noche triste"
el intérprete Edmundo Rivero.

Una interrelación es una relación o conexión que vincula a dos o más entidades, no necesariamente distintas. Por ejemplo, si a ciertos alumnos les gustan ciertos tangos y ese dato nos interesa, podemos definir una interrelación que asocia a cada alumno con 0, 1 o más tangos, y a cada tango con 0, 1 o más

alumnos. Ésta es una interrelación binaria porque asocia dos entidades, alumnos y tangos. Si queremos representar que a ciertos alumnos les gustan ciertos tangos cantados por ciertos intérpretes, podemos postular una interrelación ternaria que asocia cada alumno con 0 o más pares de un tango y un intérprete. Dijimos más arriba que las entidades que participan en una interrelación no son necesariamente distintas: por ejemplo, es bien sabido que a ciertos alumnos les gustan ciertos otros alumnos o alumnas. Si podemos obtener esta información y queremos representarla, lo hacemos mediante una interrelación binaria entre la entidad alumnos y la misma entidad alumnos.

Diagramas de Entidad-Interrelación

Es conveniente usar una notación gráfica para representar esquemas de entidad-interrelación. Cada entidad se representa mediante un rectángulo, y cada interrelación mediante un rombo. Una interrelación n-aria se conecta en el diagrama a los n rectángulos que participan en ella usando segmentos de recta. Por ejemplo, la figura 2.1 muestra algunas de las entidades e interrelaciones definidas más arriba.

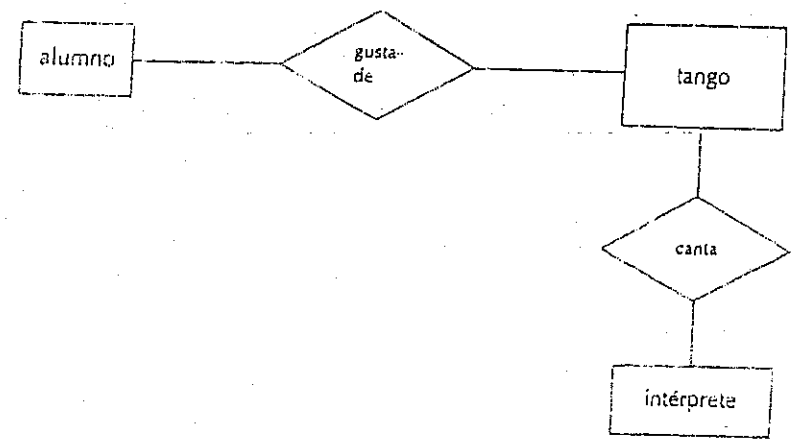


Figura 2.1
Un diagrama de entidad-interrelación

Funcionalidad de las Interrelaciones Binarias

En el caso de las interrelaciones binarias, que son probablemente las más frecuentes en muchas aplicaciones, podemos distinguir entre interrelaciones funcionales y no funcionales. Una interrelación C entre dos entidades E_1 y E_2 es funcional de E_1 a E_2 si asocia cada instancia de E_1 con a lo sumo una instancia de E_2. En términos matemáticos, esto equivale a decir que la relación binaria definida por C es una función parcial del conjunto de instancias de E_1 al conjunto de instancias de E_2. Por ejemplo, si introducimos la entidad provincia en el esquema de la Figura 2.1 y agregamos la interrelación nacido-en entre alumno y provincia, verificamos que esta interrelación es funcional, ya que cada alumno nació en a lo sumo una provincia. Observemos que puede haber alumnos no nacidos en ninguna provincia; por ejemplo, alumnos de Capital Federal, o nacidos en el Brasil. Como la definición lo indica, la funcionalidad de una interrelación depende del lado del que se la mire, nacido-en es funcional de alumno a provincia, pero no de provincia a alumno, ya que en cada provincia pueden haber nacido muchos alumnos. En nuestra notación gráfica, si la interrelación binaria C es funcional de E_1 a E_2, lo indicaremos agregando una punta de flecha al segmento que conecta a C con E_2, como en la figura 2.2. Es posible que una interrelación binaria sea funcional en ambas direcciones, aunque tales interrelaciones tienden a escasear en la vida real.

Figura 2.2
Una interrelación funcional

Atributos y Claves

Finalmente, el modelo de entidad-interrelación contiene el concepto de *atributo*. Un atributo es una propiedad de una entidad o interrelación. Cada instancia de la entidad o interrelación tendrá un valor particular para cada uno de sus atributos. Por ejemplo, la entidad alumno probablemente tendrá los atributos número de alumno, nombre, carrera, dirección, etc. Si a un

alumno le gustan mucho ciertos tangos y menos otros, podemos agregar a la interrelación gusta-oír un atributo numérico que codifique cuánto le gusta a cada alumno cada tango. En los diagramas de entidad-interrelación, los atributos se indicarán como círculos conectados a las entidades o interrelaciones a que pertenecen. Nosotros evitaremos representar los atributos gráficamente para no complicar demasiado los diagramas.

En la construcción de un modelo de entidad-interrelación, es necesario especificar cómo se distinguirán dentro de la base de datos las distintas instancias de cada entidad e interrelación. Esto se hace indicando que cierto atributo o conjunto de atributos forma una clave. Una clave es un conjunto de atributos de una entidad tal que no existen dos instancias de la entidad que tengan el mismo conjunto de valores en ese conjunto de atributos. Por ejemplo, presumiblemente los números de alumno son únicos, o sea, no existen dos alumnos con el mismo número; entonces el número de alumno es una clave para la entidad alumno. Para la entidad tango, el nombre del tango puede o no ser suficiente para identificarlo en forma única; si admitimos la posibilidad de que haya más de un tango con el mismo nombre, podemos tomar el conjunto {nombre, autor} como clave, suponiendo que ningún autor cometerá la distracción de componer dos tangos distintos con el mismo nombre. En la mayor parte de las aplicaciones es conveniente tener un solo atributo formando la clave, y no varios; en lugar de depender de la suposición de que ningún autor compone dos tangos con el mismo nombre, es más simple inventar un atributo, digamos número de tango, y asignar un único valor del atributo a cada instancia de la entidad.

En general, una entidad puede tener múltiples claves; por ejemplo, si la entidad alumno tiene el atributo número de documento de identidad además de número de alumno, cualquiera de estos dos puede servir como clave. De ahora en más vamos a suponer que el diseñador del esquema de datos especifica para cada entidad un único conjunto de atributos que forma la *clave primaria*. Por ejemplo, en el caso de la entidad alumno, el diseñador especificará si la clave primaria es el número de alumno o el número de documento.

Hemos hablado de claves de entidades; para las interrelaciones, el diseñador no necesita especificar una clave. Si la interrelación C conecta entidades $E_1, \ldots, E_n$, la unión de las n claves de las entidades formará una clave para la interrelación. Notemos que esto implica que cada conjunto de instancias $e_1, \ldots, e_n$ de las entidades $E_1, \ldots, E_n$ participa a lo sumo en una instancia de la interrelación C.

2.3. MODELO RELACIONAL

El modelo relacional introducido por E.F. Codd a principios de la década del 70, fue el primer modelo de datos explícitamente descripto como tal. Hasta ese momento, los sistemas de bases de datos existentes pertenecían a una de dos categorías: sistemas *jerárquicos*, tales como el *IMS* de IBM, y sistemas de *red* como el *IDS* de General Electric/Honeywell. Pero no existía una noción clara de modelo de datos, y los sistemas existentes eran complejos y carentes de una base conceptual sólida. El mérito del modelo relacional fue proponer una base matemática simple y bien definida para el estudio de problemas de bases de datos.

Una base de datos relacional consiste de un conjunto de *tablas*, cada una de las cuales tiene asignado un nombre único. Por ejemplo, la tabla *Libros* de la figura 2.3 tiene tres *atributos*: *autor*, *título* y *año*. Para cada atributo existe un conjunto de posibles valores, llamado el *dominio* de ese atributo. Por ejemplo, el dominio del atributo *año* es el conjunto de todos los enteros, digamos, entre 1800 y 2000. Sea D_1 el dominio del primer atributo de la tabla *autor*, y D_2, D_3 los dominios de los otros dos atributos. Entonces cada entrada en la tabla o *fila* se puede pensar como un elemento del producto cartesiano

$$D_1 \times D_2 \times D_3.$$

En matemática, una *relación* sobre n dominios se define como un subconjunto del producto cartesiano de los dominios, o sea, un conjunto de *tuplas*, cada una de las cuales se obtiene tomando un elemento de cada dominio. Esto corresponde casi exactamente a nuestro concepto de tabla, y una tupla corresponde a una fila de la tabla.

Título	Autor	Año
Ficciones	Borges	1944
Los lanzallamas	Arlt	1931
Rayuela	Cortázar	1963
Pubis angelical	Puig	1979

Figura 2.3
Una relación de rango 3

Por lo tanto, vamos a usar el término relación para referirnos a una tabla, y tupla para referirnos a una fila. Una relación definida sobre n dominios se dice de *rango n* o *n-aria*. La cantidad de tuplas en la relación es la *cardinalidad*. La relación Libros de la figura 2.3 es de rango 3 y cardinalidad 4. La cardinalidad de una relación varía con el tiempo a medida que se insertan o eliminan tuplas; el rango varía raramente o nunca.

¿Por qué hemos dicho que el concepto de relación matemática corresponde *casi* exactamente a lo que necesitamos? Porque en las relaciones matemáticas, las columnas o componentes se identifican simplemente por el orden en que aparecen, de 1 a *n*, mientras que, en las tablas de una base de datos relacional, las columnas tienen nombres que llamamos *atributos*. Como veremos más adelante, a veces resulta conveniente identificar las columnas simplemente por número, a veces por nombre, de modo que, si t es una tupla de la relación *Libros*, aceptaremos cualquiera de las dos notaciones siguientes para referirnos a la componente *autor*:

t[autor]

o bien

t[2]

Por ejemplo, si

t = <"Ficciones","Borges", "1944">,

entonces

t[2] = t[autor] = "Borges".

Notemos que hemos definido una relación como un *conjunto de tuplas*. Esto tiene varias consecuencias importantes. Primero, un conjunto no posee elementos duplicados. Por lo tanto, ninguna tupla puede aparecer más de una vez en una relación. Segundo, el orden de los elementos de un conjunto es arbitrario. O sea que el orden en que las tuplas aparecen en una relación no está definido y no tiene sentido referirse a "la primera" o "la tercera" tupla. Por último, como veremos al describir el álgebra relacional, todas las

operaciones habituales de conjuntos, como unión, intersección y diferencia, son aplicables a relaciones.

Como cualquier modelo de datos y de acuerdo con nuestra definición original, el modelo relacional se define por:

i. Objetos e interrelaciones: los objetos básicos de este modelo son las relaciones o tablas. Las interrelaciones entre dos o más tablas se definen por medio de atributos comunes o compartidos.
ii. Conjunto de operaciones: son las operaciones del álgebra relacional que presentaremos en el próximo capítulo y, como se verá más adelante, tienen una equivalencia en el cálculo relacional. Los operadores más comunes son el de selección, la proyección, la junta natural, la unión, diferencia y producto cartesiano.
iii. Restricciones: como se expresó más arriba, las relaciones no pueden tener filas duplicadas (por tratarse de un conjunto), los nombres de columnas en una tabla no pueden estar repetidos, los valores que pueden tomar los atributos deben ser atómicos, son algunas de las restricciones que podemos enumerar para este modelo.

Haremos una distinción entre relaciones y *esquemas de relación*. Un esquema de relación consiste de la lista de atributos de la relación. Por ejemplo, para la relación *Libros*, el esquema es *(Título, Autor, Año)*. Generalmente nos referiremos a un esquema de relación dando el nombre de la relación seguido de la lista de atributos entre paréntesis: *Libros (Título, Autor, Año)*. A veces se incluye también en la lista de atributos el dominio de cada atributo, pero esto no será necesario para nuestros fines. La diferencia entre esquema de relación y relación es semejante a la diferencia en un lenguaje de programación entre un tipo de datos y una variable particular de ese tipo, o sea, el esquema define la estructura de la relación independientemente de los valores particulares que pueda tomar la relación. El *esquema de base de datos* es el conjunto de todos los esquemas de relación definidos para esa base de datos; y una *instancia de base de datos* es la asignación de una relación particular a cada esquema de relación en el esquema de base de datos.

Por ejemplo, podríamos agregar a nuestra base de datos literaria dos esquemas de relación:

Nacionalidad (Autor, País), y
Ediciones (Título, Autor, Editorial, Año)

Las tuplas de *Nacionalidad* indican el país de origen de cada autor, como lo muestra la figura 2.4.

Autor	País
Borges	Argentina
Arlt	Argentina
Joyce	Irlanda
Cortázar	Argentina
Fuentes	México

Figura 2.4
La relación *Nacionalidad*

El esquema *Ediciones* lista todas las ediciones de cada obra, mientras que en la relación *Libros* aparece solamente el año de la primera edición. Observemos que los atributos *Autor* y *Título* aparecen en más de un esquema de relación. Esto no es casual; el uso de atributos comunes a dos esquemas de relaciones es útil para sugerir cómo correlacionar información de las dos relaciones. Por ejemplo, si queremos saber qué libros de autor argentino fueron publicados en 1999, seleccionamos de la relación *Ediciones* todas las tuplas cuyo valor en el atributo *año* es igual a 1999, y de éstas retenemos aquellas cuyo valor en el atributo *autor* aparece en la relación *Nacionalidad* con el valor *"Argentina"* en el atributo *País*.

Otra observación respecto de la figura 2.4 es que contiene información sobre algunos autores (Joyce, Fuentes) que no aparecen en la relación *Libros* de la figura 2.3, y esta última menciona un autor (Puig) que no aparece en la relación *Nacionalidad*. Esto nos indica que la información en esta base de datos es *incompleta*. A veces esta incompletitud es aceptable: podemos querer registrar la nacionalidad de un autor aunque no sepamos de ninguna de sus obras, en lugar de insistir en insertar toda la información a la vez. Supongamos en cambio que tenemos una base de datos de alumnos, con una relación maestra de alumnos y otra que describe en qué materias está anotado cada alumno. Sería razonable entonces aceptar alumnos que no están anotados en ninguna materia, pero probablemente querríamos requerir que todo número de alumno que aparece en la relación de inscripción tiene que aparecer también en la relación maestra.

En el modelo relacional existe también la noción de clave que presentamos para el modelo de entidad-interrelación pero ahora vamos a definirla más cuidadosamente.

- Sea R un esquema de relación y X un subconjunto de R, o sea, un conjunto de atributos. Decimos que X es una super-clave de R si ninguna instancia de R puede contener dos tuplas distintas t_1, t_2 tales que $t_1[X] = t_2[X]$

Por ejemplo consideremos el esquema

Exámenes (Alumno, Materia, Fecha, Nota)

El conjunto de atributos *{Alumno, Materia}* no forma una superclave ya que un mismo alumno puede verse obligado a rendir la misma materia más de una vez. En cambio *{Alumno, Materia, Fecha}* forma una superclave ya que un alumno rinde una materia una sola vez en una fecha dada.

La definición de superclave es algo débil: por ejemplo, el esquema de relación en su totalidad siempre forma una superclave (¿por qué?)

- Una clave es una superclave que tiene la propiedad adicional de que ningún subconjunto propiamente contenido en ella es una superclave. O sea una clave es una superclave que no contiene ningún atributo 'innecesario' para identificar una tupla en forma única.

Por ejemplo, toda superclave que consiste de un solo atributo es una clave. El conjunto *{Alumno, Materia, Fecha}* es una clave si suponemos que un mismo alumno puede dar la misma materia en fechas distintas y también puede dar más de una materia en una fecha dada. Si en cambio un alumno puede dar solamente un examen por día, el subconjunto propio *{Alumno, Fecha}* es suficiente para identificar una tupla en forma única y por lo tanto *{Alumno, Materia, Fecha}* es una superclave, pero no una clave.

Observemos que la determinación de superclaves y claves no es un procedimiento formal, sino que depende de las suposiciones que uno haga sobre el mundo exterior que se intenta representar en la base de datos. En particular, no es posible determinar qué atributos forman una clave o superclave examinando una instancia particular de una relación; simplemente porque hasta el momento nunca se han dado dos exámenes en el mismo día no es razón para concluir que esto no va a ocurrir mañana.

2.4 REPRESENTACIÓN RELACIONAL DE ENTIDADES E INTERRELACIONES

El modelo relacional se ubica a un nivel semántico mucho más bajo que el de entidad-interrelación. Es decir, las nociones de tupla, atributo y relación son independientes del mundo exterior, a diferencia de los conceptos de entidad e interrelación, que representan directamente objetos externos. Una relación puede representar una entidad, una interrelación entre entidades, una combinación de entidades e interrelaciones o ninguna de estas cosas. Podemos decir que el modelo de entidad-interrelación es al relacional como PASCAL es al lenguaje assembler: en assembler podemos escribir programas que no tienen equivalente en PASCAL, pero generalmente esto no es una buena práctica, y si nos vemos obligados a programar en assembler, trataremos de usar solamente estructuras equivalentes a if-then-else, while, etc. Del mismo modo, es recomendable limitarse a usar relaciones que representen claramente una entidad o una interrelación.

Vamos a mostrar que todo esquema de entidad-interrelación admite una representación natural en términos relacionales. Sea E una entidad con atributos $A_1, \ldots, A_n$. Para representar E usamos un esquema de relación $R_E(A_1, \ldots, A_n)$. La clave de R_E es la clave de E. Cada tupla de R_E corresponde a una instancia de la entidad E. Por ejemplo, la entidad *Alumno* con atributos *número, nombre, dirección, nacimiento* se representa mediante el esquema de relación

R_{Alumno} *(número, nombre, dirección, nacimiento)*

La clave de R_{Alumno} es *número*.

Sea C una interrelación entre las entidades $E_1, \ldots, E_n$. Sean $K_1, \ldots, K_n$ las claves primarias de las entidades. Para representar C, formamos un esquema de relación R_C. R_C tiene como atributos $K_1, \ldots, K_n$. La unión de los atributos en $K_1, \ldots, K_n$ forma la clave de R_C. Si C tiene atributos $B_1, \ldots, B_m$, agregamos estos atributos a R_C. Por ejemplo, consideremos una interrelación *Anotado* entre las entidades *Alumno* y *Carrera*. Supongamos que la clave primaria de *Alumno* es *número* y la de *Carrera* es *título*, y que *Anotado* tiene dos atributos: el año en que el alumno se anotó en la carrera, *año*, y el tipo de inscripción (libre, regular, etc.): *tipo*. El esquema de relación

$R_{Anotado}$ (*número, título, año, tipo*),

con clave *(número, título)*, representa la interrelación *Anotado*.

EJERCICIOS

E1. Un modelo de datos es, básicamente, un lenguaje que brinda una serie de estructuras con las cuales se puede expresar el conocimiento sobre una aplicación del mundo real. ¿Considera que esto es un factor limitante? ¿Existen cosas que no podrían ser modelizadas? Si el lenguaje natural es sumamente expresivo, ¿por qué no utilizarlo para modelizar datos? Si lo que se requiere es un lenguaje absolutamente formal ¿por qué no utilizar el lenguaje de la lógica?

E2. En un modelo de datos se especifican ciertas restricciones para los datos contenidos en una bd. Éstas pueden expresarse en forma declarativa (con fórmulas de la lógica de primer orden, con dependencias funcionales, con tipos de datos de algún lenguaje de programación, etc.) o bien en forma procedimental mediante un programa. ¿Qué método le parece más adecuado? Considere expresividad, eficiencia, costo en programación, simpleza, comprensibilidad, etc.

E3. Muchas veces es dificultoso distinguir entre entidades, atributos e interrelaciones cuando se intenta modelizar utilizando el Modelo ER. Enuncie algunas reglas que puedan servir como guía para distinguir cada uno de estos conceptos. Dar ejemplos en el que un mismo objeto del mundo real puede ser modelizado como entidad, interrelación y atributo.

E4. En el esquema de la figura 2.5, ¿en qué condiciones sería válido eliminar, por redundante, la interrelación A-C? Contestar a la misma pregunta si las interrelaciones A-B y B-C fueran opcionales (por ejemplo, algún elemento de A no interrelacionado con ningún elemento de B).

E5. i. ¿Es posible reemplazar la relación ternaria de la figura 2.6.a por un conjunto de relaciones binarias equivalentes? Explique cómo lo haría y justifique.
ii. ¿Cómo procedería si la relación de A-B hacia C fuera de muchos a uno como en la figura 2.6.b?

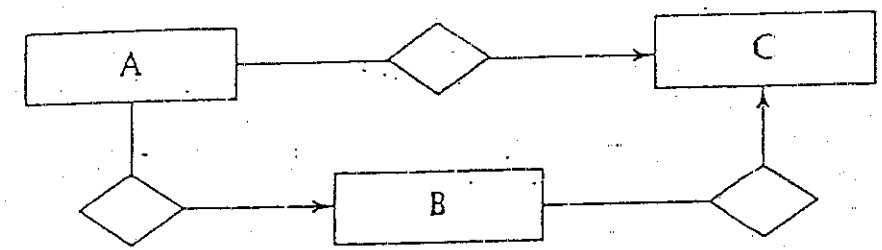


Figura 2.5

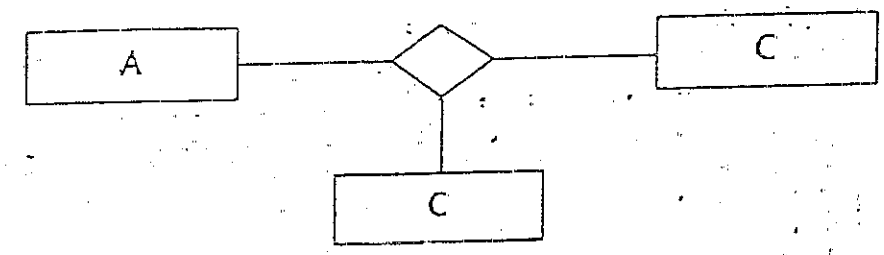


Figura 2.6.a

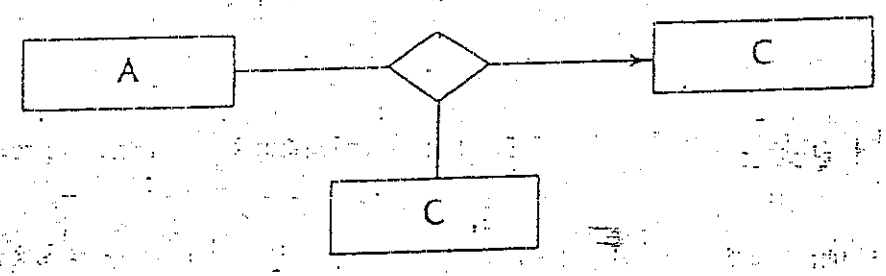
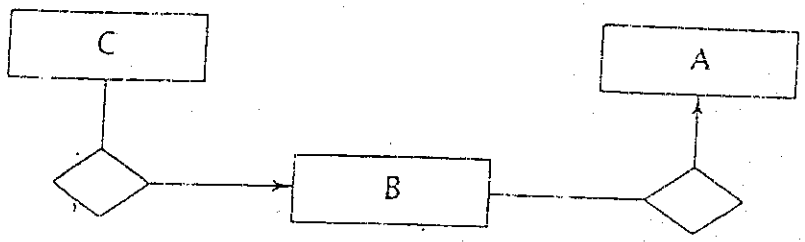


Figura 2.6.b

E6. Definir claramente los mecanismos de abstracción y dar al menos tres ejemplos en los que se aplique cada uno de ellos.

E7. Dado el siguiente Diagrama ER y las consultas:
i. hallar todos los $c \in C$ relacionados con $a_i \in A$.
ii. hallar todos los $c \in C$ relacionados con a_i y b_i ($a_i \in A$ y $b_i \in B$).
Enunciar las condiciones que deben cumplir las instancias de A, B y C para que las respuestas de i. y ii. coincidan.



E8. Modelizar la biblioteca de la facultad utilizando el Modelo ER. Existen dos tipos de usuario: sin carnet (sólo pueden realizar consultas en sala de lectura) y con carnet (pueden llevarse libros en préstamo hasta por una semana). Interesa registrar todos los préstamos. El material bibliográfico está compuesto por libros y revistas, y estas últimas sólo pueden consultarse en sala de lectura. Un libro o revista puede tener varios ejemplares.
Describir las entidades con sus correspondientes atributos, identificar las claves e interrelaciones con su clasificación. Realizar el diagrama ER.

E9. Utilice el Modelo ER para modelizar la bd de una empresa constructora. La empresa se encuentra organizada en distintos sectores, cada uno de los cuales tiene un encargado de quien dependen diversos empleados. Cada sector se corresponde con un grupo de actividades específico en una obra y puede o no estar afectado a una obra en un cierto momento. Los empleados se dividen en efectivos y eventuales. Estos últimos se contratan durante la obra de acuerdo con las necesidades de la misma, y están asignados a una obra en particular, mientras que los empleados efectivos pueden atender más de una obra simultáneamente.
Describir las entidades con sus correspondientes atributos, identificar las claves e interrelaciones con su clasificación. Realizar el diagrama ER.

E10. ¿Cómo modelizaría en el modelo relacional una relación de uno a muchos, y una de muchos a muchos? Plantear las distintas posibilidades y considerar ventajas y desventajas.
¿Cómo haría para representar en el modelo relacional las jerarquías de entidades?

E11. Traducir al modelo relacional los esquemas realizados en los ejercicios E8 y E9. Indicar las claves candidatas y seleccionar las claves primarias para cada relación. Especificar qué atributos podrían tomar valores nulos.

E12. Diseñar una base de datos para "Good Bye Tours", una empresa de viajes y turismo, utilizando el Modelo ER. Del análisis de requerimientos se ha determinado lo siguiente:

- Los productos comercializados por GBT son paquetes de servicios turísticos que se compran a operadores mayoristas. Es necesario almacenar información sobre los operadores, a saber: nombre, domicilio, localidad, país, télex, teléfono, monto de la deuda a pagar por GBT.
- Los paquetes se caracterizan por su código, itinerario, costo y precio. También se debe almacenar una descripción de cada recorrido. Periódicamente se arman contingentes de pasajeros que toman un mismo conjunto de paquetes. Los datos del contingente son nombre, nombre del guía, fecha de partida, duración, cupo máximo, cantidad de pasajeros.
- Por cada pasajero se almacena su nombre, nro. de pasaporte, nacionalidad, sexo y edad.
- Se emiten facturas a nombre de los pasajeros titulares. Cada factura tiene los datos corrientes de las facturas tipo A, B o C, según corresponda, además del vencimiento de la misma, forma de pago y la descripción de cada ítem facturado.
- Un contingente puede contratar uno o más paquetes y un paquete puede estar programado en varios contingentes en fechas diferentes.
- Cada paquete pertenece a un único operador.
- Cada pasajero participa a lo sumo de un contingente.

Describir las entidades con sus correspondientes atributos, identificar las claves e interrelaciones con su clasificación. Realizar el diagrama ER. Traducir el modelo ER al modelo relacional.

E13. Diseñar una base de datos, utilizando el Modelo ER, para el Instituto KCC, de enseñanza de computación, con el fin de sistematizar sus actividades académicas. Del análisis de requerimientos se obtuvo lo siguiente:

- Un curso se caracteriza por una sigla de 4 letras, nombre, fecha de inicio, duración, cupo y arancel.
- Cada alumno se describe por un legajo, nombre, domicilio y teléfono.
- Cada ítem de horario se representa de la forma día de la semana, horario (desde, hasta) y aula. Por ejemplo: Lu, 15, 18, 305, lo que representa el ítem lunes de 15 a 18 en el aula 305.
- Los cargos se almacenan como número de cargo, categoría y sueldo. Cada cargo pertenece a un único profesor y a un único curso, pero

un profesor puede tener varios cargos y un curso puede tener asignados varios cargos.

- Un ítem de horario corresponde a un único curso pero puede haber varios horarios para el mismo curso (uno por cada clase semanal).
- Los alumnos pueden inscribirse en varios cursos.

Describir las entidades con sus correspondientes atributos, identificar las claves e interrelaciones con su clasificación. Realizar el diagrama ER. Traducir el modelo ER obtenido al modelo relacional.

E14. Dado el siguiente esquema de base de datos relacional, obtener al menos un diagrama ER del cual se haya podido derivar.

R_1 (A1, A2), R_2 (A1, B1, B2), R_3 (C1, C2, A1), R_4 (D1, D2),

R_5 (L1, L2, L3), R_6 (C1, K1, L1), R_7 (A1, T1, E2, E3), R_8 (A1, T1, F1, F2),

R_9 (D1, C1), R_{10} (A1, A11), R_{11} (K1, K2, K3)

Los atributos subrayados representan claves de la relación.

NOTAS BIBLIOGRÁFICAS

Para un tratamiento detallado de modelos de datos, consultar D. C. Tsichritzis y P. H. Lochovsky [199]. Llevado al límite de generalidad, el tema de modelos de datos se confunde con el problema conocido en Inteligencia Artificial como representación del conocimiento; para una perspectiva ver R. J. Brachman y H. J. Levesque [45], M. Brodie J. Mylopoulos y J. Schmidt [47] y M. Brodie y J. Mylopoulos [46].

El modelo Entidad-Interrelación(ER) fue introducido por Peter Chen en P. Chen [60].

El modelo relacional fue introducido inicialmente en un artículo de E. F. Codd, "A Relational Model for Large Shared Data Bases". E. F. Codd [62].

Existen numerosos textos especializados en diseño de bases de datos que utilizan el modelo Entidad-Interrelación para la modelización a nivel conceptual, y el modelo relacional para la modelización a nivel lógico. Entre ellos se destacan las obras de C. Batini, S. Ceri y S. Navathe [17], H. Mannila y K. Raiha [157] y la de T. Teorey [198]. También son numerosos los textos especializados en bases de datos relacionales, por ejemplo: I. H. Merrett [160]; J. W. Schmidt and M. L. Brodie [189]; S. Alagic [8]; S. Mittra [165]; P. Atzeni y V. De Antonellis [13].

CAPÍTULO 3

ÁLGEBRA Y CÁLCULO RELACIONALES

Una de las funciones básicas de un sgbd es la de proveer acceso a la información a un alto nivel de abstracción. Por "acceso" entendemos en primer lugar recuperación o extracción de información, pero también por supuesto el sistema debe proveer funciones de actualización y definición o redefinición de la estructura de la base de datos. Generalmente los sgbd's empaquetan este conjunto de funciones en un lenguaje de alto nivel llamado, por abuso de terminología, *lenguaje de consulta*. El lenguaje de consulta suele ser la interfaz más visible de un sgdb.

En este capítulo discutiremos en primer lugar dos lenguajes de consulta, el álgebra relacional y el cálculo relacional, cuya función no es la de servir como herramientas prácticas de uso de una base de datos, sino como modelos teóricos de la capacidad de recuperación de información que todo lenguaje de consulta debe poseer como mínimo. Mostraremos que estos dos lenguajes, en apariencia muy distintos, son equivalentes en el sentido que cualquier consulta expresable en uno de ellos es también expresable en el otro. Dada la importancia conceptual de estos lenguajes, se ha dado en llamar *relacionalmente completo* a cualquier lenguaje de consulta que tenga el mismo poder expresivo que el álgebra y cálculo relacionales.

Como mencionamos más arriba, no basta con proveer facilidades de extracción de la información sino que un lenguaje práctico debe también permitir definir la estructura inicial de la base de datos, modificar dicha estructura, actualizar los datos, y otras funciones auxiliares que veremos más adelante. En el capítulo próximo presentaremos en detalle un lenguaje llamado SQL que provee todas estas funciones.

3.1. ALGEBRA RELACIONAL

El álgebra relacional consiste de un conjunto de *operadores* que se aplican a una o dos relaciones y producen como resultado otra relación. Una consulta se formaliza en este lenguaje como una expresión compuesta de operadores, variables relacionales que representan los operandos, y constantes.

Vamos a utilizar un ejemplo para ir presentando los operadores y su semántica. Consideremos la relación *hospital* de la Figura 3.1.

Código	Nombre	Camas
22	Rivadavia	412
13	Central	502
45	De Niños	845
18	Italiano	987

Figura 3.1
Relación *Hospital*

Operadores Unarios

El primer operador que vamos a presentar es la *selección* σ. Éste es un operador unario, o sea, toma como operando una sola relación, y su valor es cierto subconjunto de las tuplas de esa relación. Por ejemplo, si queremos saber qué hospitales tienen más de 800 camas, la expresión

$$\sigma_{camas > 800}\ (Hospital)$$

producirá:

Código	Nombre	Camas
45	De Niños	845
18	Italiano	987

Para determinar todas las tuplas cuyo código es igual a 22, la expresión:

$$\sigma_{código = 22}\ (Hospital)$$

producirá:

Código	Nombre	Camas
22	Rivadavia	412

Notemos que el mismo resultado sería producido por la expresión

$$\sigma_{nombre = Rivadavia}\ (Hospital)$$

Para ver un caso donde la condición de selección es más compleja, agreguemos a nuestra base de datos la relación *Guardia* que muestra la figura 3.2. Cada guardia pertenece a cierto hospital, identificado por su código, tiene un número de guardia, un nombre y una cantidad de camas. Supongamos que queremos extraer las tuplas correspondientes a guardias de maternidad o cuidado intensivo que no son del hospital número 13 y tienen entre 20 y 40 camas; la selección correspondiente será:

$$\sigma_{hosp \neq 13\ \&\ (nombre = CuidInt\ |\ nombre = Mat)\&camas \geq 20\&camas \leq 40}\ (Hospital)$$

y el valor de esta expresión es:

Hosp	Número	Nombre	Camas
22	6	Infecc.	60
22	6	CuidInt	21
22	2	Mat	35
13	1	Mat	25
13	2	CuidInt	40
13	3	Emerg	50
45	1	Mat	120

Figura 3.2
Relación *Guardia*

y S de rango r y s respectivamente, el producto cartesiano $R \times S$ es una relación de rango $r + s$ conteniendo todas las tuplas cuyas r primeras columnas forman una tupla que aparece en R y cuyas s restantes columnas forman una tupla que aparece en S. En general, el resultado de un producto cartesiano no será de interés en sí mismo, sino que le aplicaremos otros operadores para obtener la respuesta deseada. Por ejemplo, supongamos que queremos saber quiénes son los pediatras que trabajan en el hospital Rivadavia. Sin saber cuál es el código del hospital Rivadavia, no podemos extraer esta información de la relación *Médico* por sí sola. Tomamos en cambio el producto cartesiano de *Médico* con *Hospital*, esto nos dará todas las tuplas posibles que se obtienen concatenando una tupla de *Hospital* con una de *Médico*. Luego seleccionamos de esta relación aquellas donde los códigos de hospital coinciden, la especialidad es pediatría, y el nombre del hospital es Rivadavia, y finalmente proyectamos sobre la columna 3, o sea, el nombre de médico:

$$\pi_3\,(\sigma_{1=5\ \&\ 4=\text{pedia}\ \&\ 6=\text{Rivadavia}}\,(\textit{Médico} \times \textit{Hospital}))$$

Observemos que podríamos haber obtenido el mismo resultado haciendo dos de las selecciones antes de aplicar el producto:

$$\pi_3\,(\sigma_{1=5}(\sigma_{4=\text{pedia}}\,(\textit{Médico}) \times \sigma_{2=\text{Rivadavia}}\,(\textit{Hospital})))$$

Operadores de Junta

La familia de operadores de *junta* permite expresar en forma más concisa la combinación de un producto cartesiano seguido de una selección. Dada una relación R de rango r y una relación S de rango arbitrario,

$$R \underset{i\,\theta\,j}{|\times|} S$$

donde θ es uno de los comparadores $=, \neq, <, >, \leq, \geq$, i es un número de columna de R, y j es un número de columna de S, es equivalente a

$$\sigma_{i\,\theta\,r+j-1}\,(R \times S)$$

Por ejemplo, la consulta sobre los pediatras del hospital Rivadavia mencionada más arriba puede escribirse usando junta como

$$\pi_3\,(\sigma_{4=\text{pedia}}\,(\textit{Médico}) \underset{1=1}{|\times|} \sigma_{2=\text{Rivadavia}}\,(\textit{Hospital}))$$

Finalmente, el operador de *junta natural* es aplicable solamente cuando las columnas tienen nombre. Sea R una relación definida sobre el conjunto de atributos A_R y S sobre A_S. La junta natural de R y S es una relación definida sobre atributos $A_R \cup A_S$ por la siguiente fórmula:

$$R\ |\times|\ S = \{t \,/\, \pi_{A_R}(t) \in R \text{ y } \pi_{A_S}(t) \in S\}$$

Por ejemplo, sean R y S las dos relaciones siguientes:

A	B	C
0	1	1
0	2	1
1	1	2
1	3	2

B	C	D
1	1	4
1	1	5
1	2	6
3	3	7

Entonces $R\ |\times|\ S$ es:

A	B	C	D
0	1	1	4
0	1	1	5
1	1	2	6

Notemos que la junta natural es definible como un producto cartesiano seguido de una selección y proyección; en el ejemplo de más arriba

$$R \mid\times\mid S = \pi_{ABCD}\,(\sigma_{B=B\&C=C}\,(R \times S))$$

Finalmente, señalamos que, en el caso especial en que $A_R \cap A_S = \emptyset$, la junta natural se reduce al producto cartesiano de R y S.

Operador de División o Cociente

Ésta es una operación que, en cierta forma, resulta la inversa del producto cartesiano. Dadas dos relaciones R y S ($S \neq \emptyset$) cuyos esquemas sean $R = A_1, \ldots, A_p, B_1, \ldots, B_q$ y $S = B_1, \ldots, B_q$, se define R / S (R dividido por S) como la relación T que tiene atributos $A_1, \ldots, A_p$, con el mayor conjunto posible de tuplas y que cumple que $T \times S \subseteq R$: cada tupla de T verifica que concatenada con toda tupla de S da alguna tupla de R.

Veamos cómo se puede expresar la división en función de los operadores anteriores. Las tuplas de T se podrían obtener proyectando las primeras p columnas

$$\pi_{1, \ldots, p}\,(R)$$

Llamemos P a la relación recién obtenida. Veamos ahora cuáles tuplas de P no cumplen con la condición impuesta a las tuplas de T. La expresión sería la siguiente:

$$N = \pi_{1 \ldots p}\,(P \times S - R)$$

Luego las tuplas de P que sí cumplen son:

$$T = P - N$$

que en una única expresión resulta:

$$T = \pi_{1 \ldots p}\,(R) - \pi_{1 \ldots p}\,(\pi_{1 \ldots p}\,(R) \times S - R)$$

La definición de T podría reescribirse como:

$$T = \{t \,/\, t \in \pi_{1 \ldots p}\,(R) \,/\, (\forall u \in S\,(t\,u \in R))\}$$

Veamos un ejemplo. La relación Proveedor nos dice qué proveedor provee cuál producto mientras que la relación Selección nos da una lista de productos seleccionados.

Proveedor:

Nombre	Producto
Luis	p1
Luis	p2
Luis	p3
Luis	p4
Luis	p5
José	p2
José	p3
Alberto	p2
Alberto	p3
Alberto	p5
Juan	p1
Juan	p2
Juan	p5

Selección:

Producto
p2
p3
p5

Dada la consulta *¿quiénes son los proveedores que proveen todos los productos incluidos en Selección?*, la respuesta es la tabla

T:

Nombre
Luis
Alberto

Otro ejemplo sería *¿cuáles son los médicos que trabajan en todos los hospitales?* La siguiente expresión produce la respuesta deseada:

$$\pi_{hosp\ nombre}\,(Médico) \,/\, \pi_{código}\,(Hospital).$$

3.2. CÁLCULO RELACIONAL

En el álgebra relacional, una consulta se construye componiendo operadores que se aplican a relaciones y producen nuevas relaciones. El lenguaje que vamos a presentar a continuación tiene un estilo muy distinto: una consulta consiste en una *especificación*, formalizada en lógica de primer orden, de todas las tuplas que deseamos extraer de la base de datos. La forma general de una consulta será:

$$\{x_1, \ldots, x_n \mid \phi(x_1, \ldots, x_n)\}$$

El valor de esta consulta es una relación: el conjunto de todas las tuplas $<x_1, \ldots, x_n>$ de rango n que satisfacen la fórmula lógica ϕ. Como los x_i's representan valores de componentes de relaciones, esta versión del cálculo relacional se llama *cálculo de dominios*. Existe otra versión llamada *cálculo de tuplas* en la que las variables representan tuplas, no componentes de éstas. Es fácil demostrar que el cálculo de tuplas y el cálculo de dominios pueden expresar exactamente la misma clase de consultas; en este trabajo nos limitaremos a tratar el cálculo de dominios.

Veamos algunos ejemplos antes de dar la forma general de la fórmula ϕ. Una vez más, los hospitales con más de 800 camas:

$$\{c\ n\ m \mid Hospital\ (c\ n\ m)\ \&\ m > 800\}$$

Esto se interpreta como "el conjunto de todas las triplas $<c,n,m>$ tales que $<c,n,m>$ aparece en la relación *Hospital* y m es mayor que 800". Recordemos que esta consulta se expresa mediante una selección en el álgebra relacional. La consulta "personas que se aman *a* sí mismas" es particularmente simple:

$$\{a \mid Ama\ (a, a)\}$$

Esto es equivalente a una selección seguida de proyección, $\pi_1(\sigma_{1=2}(Ama))$. ¿Cómo obtenemos la lista de las personas que aman a alguien distinto de sí mismas? Esto equivale al conjunto de personas *a* tales que *existe* una persona *c* tal que *a* ama a *c* y *c* no es *a*; para expresarlo formalmente, usamos el *cuantificador existencial* $\exists$:

$$\{a \mid (\exists\ c)\ Ama\ (a, c)\ \&\ a \neq c\}$$

Los operadores booleanos unión y diferencia son fácilmente implementables usando disyunción y conjunción seguida de negación. Por ejemplo, la lista de nombre y número de hospital de todo el personal, médico y no médico, se expresa como:

$$\{n\ c \mid ((\exists\ d\ e)(Médico\ (c\ d\ n\ e)) \vee ((\exists\ g\ r)\ (Personal\ (c\ g\ n\ r))\}$$

Las personas que no se aman a sí mismas:

$$\{a \mid (\exists\ b)\ (Ama(a, b)\ \&\ \neg Ama(a,a))\}$$

donde "¬" es el operador de negación. Observemos que hemos evitado usar la expresión más simple

$$\{a \mid \neg Ama(a,a)\}$$

El motivo es que esta expresión denota el conjunto de todos los valores de *a* tales que $<a,a>$ no aparece en la relación *Ama*, pero no determina de qué dominio extraemos los valores *a*. Además de las personas que no se aman a sí mismas, esta expresión denotaría también el conjunto, digamos, de todos los paraguas.

Veamos cómo formular una consulta que en el álgebra requiere junta o producto cartesiano: los pediatras del hospital Rivadavia.

$$\{p \mid (\exists\ h\ n\ m)\ Hospital\ (h, \text{'Rivadavia'}, m)\ \&\ Médico\ (h, n, p, \text{'pedia'})\}$$

El hecho de que la misma variable h aparece en *Hospital* y en *Médico* hace que esta consulta se comporte como una junta por igualdad sobre el primer atributo, seguida de una proyección. Finalmente supongamos que un médico puede trabajar en más de un hospital y queremos saber quiénes son los médicos que trabajan en *todos* los hospitales representados en la relación *Hospital*. Para esto utilizamos el cuantificador universal $\forall$ que se lee "para todo":

$$\{p \mid (\exists\ h\ n\ e)\ Médico\ (h, n, p, e)\ \&$$

$$(\forall\ s\ m\ c)(Hospital\ (s, m, c) \rightarrow Médico\ (s, n, p, e))\}$$

El símbolo "→" es la implicación booleana; $\alpha \rightarrow \beta$ se lee "α implica β" y la implicación es cierta exactamente cuando α es falsa o β es cierta; o sea que $\alpha \rightarrow \beta$ es equivalente a $(\sim\alpha) \mid \beta$. La consulta puede parafrasearse entonces como "el conjunto de todos los nombres de médico p tales que para toda tupla de la relación *Hospital*, p aparece asociado a ese hospital en la relación *Médico*". Notemos la suposición implícita en la consulta de que un mismo médico practica la misma especialidad en todos los hospitales en que trabaja; dejamos como ejercicio modificar la consulta en caso de que esta suposición no sea cierta.

Pasemos ahora a definir formalmente el cálculo relacional. Una expresión del cálculo es de la forma $\{x_1 \ldots x_n / \phi(x_1, \ldots, x_n)\}$ donde $x_1 \ldots x_n$ son *variables de dominio*, o sea, cada x_i representa un componente de una tupla. La fórmula ϕ se compone de átomos y operadores lógicos.

Un átomo es de una de las dos formas:

(a) $R(y_1, y_2, \ldots, y_k)$ donde R es una relación de rango k y cada y_i es una constante o una variable de dominio, o

(b) $x\ \theta\ y$, donde x e y son constantes o variables de dominio y θ es un comparador.

El átomo $R(y_1, \ldots, y_k)$ afirma que los valores de aquellos y_i's que son variables deben elegirse de modo que $\langle y_1, \ldots, y_k \rangle$ sea una tupla de R. El átomo $x\ \theta\ y$ afirma que los valores de x e y deben satisfacer la relación de igualdad o desigualdad $x\ \theta\ y$.

Para definir los operadores del cálculo, es conveniente definir al mismo tiempo la noción de ocurrencias *libres* y ocurrencias *ligadas* de una variable. Esto resultará ya conocido a quienes hayan visto lógica de primer orden anteriormente. Intuitivamente, una variable ocurre *libre* en una fórmula si es necesario asignarle un valor para determinar si la fórmula es o no cierta; la variable ocurre *ligada* si está sujeta a un cuantificador existencial o universal, de modo que la fórmula en que aparece debe cumplirse para todo valor o para algún valor de la variable.

Definimos una fórmula recursivamente como sigue:

1. Todo átomo es una fórmula. Toda variable que aparece en el átomo ocurre libre en dicha fórmula.
2. Si α y β son fórmulas, también lo son $\alpha \,\&\, \beta$, $\alpha \mid \beta$ y $\sim\beta$, que *afirman*, respectivamente, que tanto α como β son ciertas, que una o las dos de ellas son ciertas y que β no es cierta. Al combinar dos fórmulas usando & o |, o negar una fórmula usando ~, el conjunto de ocurrencias libres y ligadas de las variables no se modifica.
3. Si α es una formula, también lo es $(\exists x)(\alpha)$. Toda ocurrencia de x libre en α se convierte en una ocurrencia ligada en $(\exists x)(\alpha)$. Esta fórmula afirma que existe algún valor de x tal que, si reemplazamos toda ocurrencia de x libre en α por este valor, la fórmula que resulta es cierta.
4. Si α es una fórmula, también lo es $(\forall x)(\alpha)$. Esta fórmula es equivalente a $\sim(\exists x)(\sim\alpha)$.

Una expresión en el cálculo relacional es de la forma

$$\{x_1 \ldots x_n / \phi(x_1, \ldots, x_n)\}$$

donde $x_1, \ldots, x_n$ son las únicas variables que ocurren libres en ϕ.

3.3. EQUIVALENCIA ENTRE EL ÁLGEBRA Y EL CÁLCULO

Hemos definido dos lenguajes aparentemente muy distintos que permiten especificar consultas sobre una base de datos relacional. Uno de ellos, el cálculo relacional, es básicamente el uso de la lógica matemática de primer orden, o cálculo de predicados, como herramienta descriptiva de los datos que queremos extraer. El cálculo de predicados tiene propiedades que lo hacen fundamental en la lógica matemática y es, por lo tanto, natural aceptar este lenguaje como base para la construcción de un lenguaje de consulta. Además, es relativamente fácil verificar que cualquier consulta escrita en álgebra relacional puede ser expresada en el cálculo. Basta para esto con demostrar que cada uno de los operadores algebraicos básicos puede expresarse mediante una fórmula lógica. En cambio, el conjunto de operadores algebraicos que hemos definido parece bastante arbitrario y no es obvio que no necesitemos operadores adicionales si queremos expresar, en cierto sentido "todas" las consultas. Sin embargo, E.F. Codd demostró, al proponer el álgebra y el cálculo, que estos dos lenguajes son en realidad equivalentes; en particular, que para cada consulta en cálculo relacional existe una consulta equivalente en el álgebra.

En lugar de demostrar formalmente este teorema, veamos un ejemplo que sugiere el método general para convertir una expresión en el cálculo a su equivalente algebraico. Sean R y S dos relaciones binarias y supongamos que queremos expresar el conjunto de tuplas de R tales que su primera componente no aparece como primera componente de ninguna tupla de S:

$$\{wx \mid R(wx) \;\&\; (\forall y) \neg S(wy)\}.$$

La idea de la traducción de cálculo a álgebra es la siguiente. Como primer paso, vamos a eliminar de la fórmula dada el uso de & y ∀; la fórmula que resulte será a veces menos legible, pero cuanto más limitado el repertorio de operaciones usadas en la fórmula lógica, más simple será traducirla mecánicamente a forma algebraica. Usando los hechos que $\alpha \;\&\; \beta$ es equivalente a $\neg(\neg\alpha \mid \neg\beta)$, y $(\forall x)\,\phi$ es equivalente a $\neg(\exists x)\neg\phi$, y simplificando, obtenemos:

$$\{wx \mid \neg(\neg R(wx) \mid (\exists y) S(wy))\}.$$

Tomemos ahora las subfórmulas más simples de la fórmula dada: en este ejemplo, $R(wx)$ y $S(wy)$. Es obvio que existen expresiones algebraicas equivalentes a

$$\{wx \mid R(wx)\}$$

y a

$$\{wy \mid S(wy)\}.$$

Éstas son, respectivamente, las expresiones R y S. Tratemos ahora de extender y combinar estas expresiones formando expresiones más complejas, basándonos en la forma en que las respectivas subfórmulas se combinan en la fórmula dada. Consideremos la expresión

$$\{w \mid (\exists y)\, S(wy)\}.$$

Claramente el cuantificador existencial corresponde a una proyección o sea que esta consulta es equivalente a $\pi_1(S)$.

¿Cómo modificamos la expresión algebraica R para obtener una expresión equivalente a

$$\{wx \mid \neg R(wy)\}?$$

Esta consulta lógica no está bien definida, ya que representa todas las tuplas que no aparecen en R sin especificar de qué dominio se obtienen sus valores. A los fines de traducirla al álgebra vamos a interpretarla como el resultado de tomar primero el conjunto de todas las tuplas binarias que se pueden obtener combinando valores de la base de datos y luego usar el operador de diferencia para eliminar de este conjunto aquellas tuplas que aparecen en R. Sea D la expresión algebraica que calcula el conjunto de todos los valores que aparecen en cualquier posición en alguna tupla de la base de datos:

$$D = \pi_1(R) \cup \pi_2(R) \cup \pi_1(S) \cup \pi_2(S).$$

$D \times D$, o D^2, denota entonces todas las tuplas binarias formables con valores de R y de S, y la expresión que necesitamos es:

$$D^2 - R.$$

Para obtener la expresión algebraica equivalente a

$$\{wx \mid \neg R(wx) \mid (\exists y)\, S(wy)\}$$

tenemos que combinar la expresión que nos da todas las tuplas que no están en R con la que nos da la primera columna de S. Hacemos esto tomando la unión de todas las tuplas que no están en R con todas las tuplas posibles cuya primera componente aparece en la primera columna de S:

$$(D^2 - R) \cup (\pi_1(S) \times D).$$

Finalmente, agregamos el operador de negación más exterior mediante una nueva resta de D^2:

$$D^2 - ((D^2 - R) \cup (\pi_1(S) \times D)).$$

Usando la identidad $A - (B \cup C) = (A - B) \cap (A - C)$, obtenemos

$$(D^2 - (D^2 - R)) \cap (D^2 - (\pi_1(S) \times D)).$$

Como $A - (A - B) = A \cap B$, y $D^2 \cap R = R$, simplificamos a:

$$R \cap (D^2 - (\pi_1(S) \times D)).$$

Y finalmente, dado que $R \subseteq D^2$, esto es equivalente a

$$R - (\pi_1 (S) \times D)$$

O sea, la expresión que elimina de R aquellas tuplas que no aparecen en el producto cartesiano de $\pi_1 (S)$ y D, pero una tupla de R aparece en este producto cartesiano si y sólo si su primera componente aparece en la primera columna de S, y por lo tanto la expresión produce las tuplas de R cuya primera componente no aparece en la primera columna de S, que es exactamente lo que queríamos.

3.4. Completitud Relacional

Basándose en la importancia y poder del cálculo de predicados de primer orden y en la equivalencia esbozada en la última sección entre el álgebra y el cálculo, Codd propuso denominar "relacionalmente completo" a todo lenguaje de consulta que puede expresar todas las consultas expresables en álgebra o cálculo. Es natural, sin embargo, preguntarse: ¿existen consultas útiles que no son expresables en ninguno de estos lenguajes? La respuesta es afirmativa, como lo muestra el ejemplo siguiente.

Consideremos la relación *AA (desde, hacia)* que contiene pares de ciudades tales que existen vuelos directos de Aerolíneas Argentinas de la primera a la segunda. Supongamos que queremos saber qué ciudades son accesibles desde Córdoba volando exclusivamente en Aerolíneas Argentinas, ya sea en un vuelo directo o una sucesión de cualquier cantidad de vuelos. Podríamos expresar esta consulta con la siguiente fórmula lógica:

$$\{c \,/\, AA\ (Córdoba,\ c) \mid (\exists\, c_1, \ldots c_n)$$

$$(AA\ (Córdoba,\ c_1)\ \&\ AA\ (c_1, c_2)\ \&\ \ldots\ AA\ (c_n,\ c))\}$$

Pero esta fórmula no pertenece al cálculo relacional, ya que éste no permite una cantidad indeterminada n de variables y conjunciones. Es posible demostrar que no existe ninguna fórmula en el cálculo relacional que exprese esta consulta, salvo que estemos dispuestos a aceptar una cota sobre la cantidad de ciudades intermedias.

Es un tema de interés actual en investigación la extensión de los lenguajes de consulta relacionales para poder expresar estas y otras consultas que están fuera del alcance del álgebra y el cálculo.

Ejercicios

E1. Sean R y S dos esquemas de relación con X el conjunto de atributos comunes y $Z \subseteq X$. Se cumple que:

$$\pi_Z (R \mid\times\mid S) \subseteq \pi_Z (R) \mid\times\mid \pi_Z (S)$$

Construya un ejemplo especificando una instancia de R y S, y el conjunto de atributos Z, de manera tal que se cumpla la inclusión estricta.

E2. Dados R, S y T, probar que:

i $R \mid\times\mid S = S \mid\times\mid R$
ii $(R \mid\times\mid S) \mid\times\mid T = R \mid\times\mid (S \mid\times\mid T)$
iii $(R \mid\times\mid S) / S = R$, si R y S no tienen atributos en común.

Las siguientes consultas se deberán realizar en Álgebra Relacional (AR) o Cálculo Relacional de Dominios (CRD), según se indique.

E3. Dado el siguiente esquema de base de datos relacional

Almacén (Nro, Responsable)
Artículo (Cód_art, Descripción, Precio)
Material (Cód_mat, Descripción)
Proveedor (Cód_prov, Nombre, Domicilio, Ciudad)
Tiene (Nro, Cód_art)
Compuesto_por (Cód_art, Cód_mat)
Provisto_por (Cód_mat, Cód_prov)

Realizar las siguientes consultas:

i) Listar los nombres de los proveedores de la ciudad de La Plata (AR, CRD).
ii) Listar los materiales, código y descripción, provistos por proveedores de la ciudad de Rosario (AR).

iii) Listar códigos y descripciones de los artículos compuestos por al menos un material provisto por el proveedor López (AR, CRD).
iv) Hallar los códigos y nombres de proveedores que proveen al menos un material que se usa en algún artículo cuyo precio es mayor que $100 (AR, CRD).
v) Hallar el o los códigos de los artículos de mayor precio (AR).
vi) Listar los números de almacenes que tienen todos los artículos que incluyen el material con código 123 (AR).
vii) Listar los códigos y descripciones de los artículos que tienen al menos un material que no provee ningún proveedor (AR, CRD).

E4. Sean los siguientes esquemas de relación:

Vuelo (Nro_vuelo, Desde, Hasta)
Avión_utilizado (Nro_vuelo, Tipo_avión, Nro_avión)
Info_pasajeros (Nro_vuelo, Documento_identidad, Nombre, Origen, Destino)

Los vuelos no pueden tener más de dos escalas y no hay cambio de tipo de avión para un mismo vuelo.

Realizar las siguientes consultas:

i) Hallar los números de vuelo desde el origen A hasta el destino F (AR).
ii) Hallar los tipos de avión que no son utilizados en ningún vuelo que pase por B (AR, CRD).
iii) Hallar los pasajeros y números de vuelo para aquellos pasajeros que viajan desde A a D pasando por B (AR, CRD).
iv) Hallar los tipos de avión que pasan por C (AR).
v) ¿Cómo respondería i) en CRD y AR si no existiera restricción sobre el número de escalas?

E5. Dada la siguiente base de datos relacional

Persona (Tipo_doc, Nro_doc, Nombre, Dirección, Fecha_Nac, Sexo)
Progenitor (Tipo_doc, Nro_doc, T_doc_hijo, N_doc_hijo)

Realizar las siguientes consultas:

i) Hallar para una persona dada, por ejemplo José Pérez, los tipos y números de documento, nombres, dirección y fecha de nacimiento de todos sus hijos (AR, CRD).
ii) Hallar para cada persona los tipos y números de documento, nombre, domicilio y fecha de nacimiento de:
   a) Todos sus hermanos, incluyendo medios hermanos (AR, CRD).
   b) Su madre (CRD).
   c) Su abuelo materno (CRD).
   d) Todos sus nietos (AR).
iii) Hallar todos los descendientes de Adán.

E6. Dada la siguiente base de datos

Proveedor (Nro_p, Nom_p, Categoría, Ciud_p)
Artículo (Nro_a, Descripción, Ciud_a, Precio)
Cliente (Nro_c, Nom_c, Ciud_c)
Pedido (Nro_p, Nro_a, Nro_c, Cantidad, Precio_total)

Responder a las siguientes consultas en Álgebra Relacional:

i. Hallar el código (nro_p) de los proveedores que proveen el artículo a146.
ii. Hallar los clientes(nom_c) que solicitan artículos provistos por p015.
iii. Hallar los clientes que solicitan algún ítem provisto por proveedores con categoría mayor que 4.
iv. Hallar los ítem pedidos por clientes de Rosario.
v. Hallar los pedidos en los que un cliente de Rosario solicita artículos producidos en la ciudad de Mendoza.
vi. Hallar los pedidos en los que el cliente c23 solicita artículos no solicitados por el cliente c30.
vii. Hallar los pares de ciudades en la forma (ciudad1, ciudad2), tales que un proveedor en ciudad1 provea artículos pedidos por clientes de ciudad2.
viii. Hallar los nombres de los proveedores cuya categoría sea mayor que la de todos los proveedores que proveen el artículo "cuaderno".
ix. Hallar los clientes que han pedido dos o más artículos distintos.

E7. Dada la base de datos

Frecuenta (nombre_persona, nombre_bar)
Sirve (nombre_bar, nombre_cerveza)
Gusta (nombre_persona, nombre_cerveza)

Usar el Álgebra Relacional para hallar las personas que:

i. Frecuentan un bar que sirve al menos una cerveza que les gusta
ii. Beben en el mismo bar que las personas a las que les gusta la cerveza "Quilmes".
iii. Beben en el mismo bar que aquellas a las que gusta una marca de cerveza que sirva dicho bar y que le guste a Charlie García.
iv. Frecuentan solamente bares que sirven alguna cerveza que les guste (asumir que cada persona frecuenta al menos una bar y le gusta al menos una cerveza)
v. No frecuentan ningún bar que sirva alguna cerveza que les guste.
vi. Frecuentan solamente los bares que sirven todas las cervezas que les gustan.
vii. Frecuentan solamente los bares que no sirven ninguna de las cervezas que no les gusta.

NOTAS BIBLIOGRÁFICAS

Ullman [202] y Abiteboul, Hull, Vianu [2] son excelentes textos para este capítulo. Ambos también tratan la relación entre la lógica matemática y las bases de datos, con extensiones de los lenguajes relacionales que eliminan las limitaciones mencionadas al final del capítulo. Ver además los libros de P.M.D. Gray [117], H. Gallaire, J. Minker [98], H. Gallaire, J. Minker, J. Nicholas [99] y Gallaire, Minker, Nicholas [101]

El álgebra relacional fue propuesta por E. Codd [63], quien además demostró la equivalencia del álgebra con el cálculo relacional de tuplas en Codd [65]. A. Pirotte [173] clasificó los lenguajes del cálculo en CRT (Cálculo Relacional de Tuplas) y CRD (Cálculo Relacional de Dominios)

Se han propuesto numerosas extensiones al álgebra relacional. Una de ellas es la de T. Merret [162]. Otra extensión significativa, en la que se incluyen operadores agregativos, es la de A. Klug [139].

CAPÍTULO 4

EL LENGUAJE DE CONSULTA SQL*

En el capítulo anterior presentamos dos lenguajes, el álgebra y cálculo relacionales, que sirven como patrón de medida para evaluar el poder de recuperación de información de cualquier otro lenguaje. Como allí dijimos, estos dos lenguajes no se proponen como herramientas prácticas de acceso a los datos, por tres razones:

- Sintaxis: su sintaxis es estilizada y concisa y resultará natural solamente a usuarios de orientación matemática.
- Poder de cómputo: aunque teóricamente "completos" en cuanto a su poder de extracción de datos, el álgebra y el cálculo carecen de construcciones que permitan calcular resultados a partir de los valores existentes en la base de datos. Por ejemplo, consultas tales como "cuántos alumnos rindieron Análisis I" o "cuál fue el examen más reciente de Álgebra" o "listar el promedio de notas para cada materia" no son expresables en estos lenguajes.
- Actualizaciones: ni el álgebra ni el cálculo incluyen funciones de actualización de los datos, obviamente necesarias en cualquier lenguaje práctico. Es más, existen numerosas funciones, que llamaremos administrativas, que no son ni de consulta ni de actualización, tales como:

  - definición del esquema de datos
  - definición de restricciones de integridad

* Se ha adoptado como convención no escribir acentos en las expresiones SQL a fin de evitar confusiones y errores en la ejecución de los comandos.

- autorización de acceso a relaciones
- especificación de *vistas*
- control de transacciones

En la práctica, los lenguajes de consulta incorporan todas estas funciones bajo una misma interfaz lingüística.

En este capítulo nos apartaremos de nuestra regla general de no ocuparnos de sistemas particulares y trataremos en cierto detalle el lenguaje de consulta SQL, por dos razones. En primer lugar, es difícil discutir lenguajes en general sin dar un ejemplo concreto de sintaxis y semántica. En segundo lugar, aunque el SQL fue originalmente diseñado por IBM e incorporado en varios de sus productos, se ha convertido en la norma de facto en tanto que prácticamente todos los sistemas relacionales, de cualquier proveedor, implementan alguna versión de SQL. Tanto la organización estadounidense de normalización, ANSI, como la internacional, ISO, han desarrollado sucesivas definiciones de un SQL estándar. Esta presentación se ajusta al estándar SQL-92. En el apéndice se describe someramente el futuro nuevo estándar, denominado SQL:1999, en estos momentos (agosto de 1999) en proceso de aprobación por ANSI e ISO.

4.1. INTRODUCCIÓN

El lenguaje SQL, bajo su nombre original de SEQUEL (Structured English Query Language), fue diseñado en la década del 70 como parte del proyecto de investigación Sistema R de IBM, uno de los primeros intentos de construir un sistema relacional de gran escala. Vamos a presentar en orden las siguientes funciones del lenguaje:

- Definición de Datos
- Consulta
- Actualización
- Definición de Vistas
- Control de Seguridad

Esto nos servirá, de paso, para introducir dos aspectos importantes de la gestión de bases de datos que no hemos tenido ocasión de tratar hasta ahora, y que no se limitan a SQL: el concepto de vista de un esquema relacional, y los requerimientos de autorización y control de acceso a relaciones.

Debemos advertir que nuestra descripción del lenguaje no será completa; presentaremos el grueso de las funciones, dejando de lado numerosos detalles. En particular, omitiremos las funciones relativas a aspectos físicos de la implementación de relaciones, tales como creación y modificación de índices.

4.1.1 Definición de Datos

Los tres verbos create, alter y drop permiten respectivamente crear un esquema de relación, modificarlo o eliminarlo. Para crear un esquema de relación, es necesario especificar el nombre de la relación y los nombres y tipos de dato de cada atributo.

EJEMPLO 4.1: la siguiente sentencia SQL define una relación llamada "Hospital", con atributos "numero", "nombre", "direcc", "telef", y "camas". El atributo "telef" es un entero de 32 bits, "nombre" es una cadena de 50 caracteres, "numero" es un entero de 16 bits. No se admiten valores nulos en el atributo "numero".

```
create table Hospital
        (numero    smallint not null,
        nombre     char (30),
        direcc     char (50),
        telef      int,
        camas      smallint);
```

El efecto del create es la creación de este esquema de relación, apropiadamente reflejada en el diccionario de datos. La relación apenas creada no contiene ninguna tupla; veremos más adelante cómo insertar tuplas en una relación.

Supongamos que queremos agregar a la relación Hospital un nuevo campo, "Director", que contendrá el nombre del director del hospital; podemos hacerlo de la siguiente manera:

EJEMPLO 4.2:

```
alter table   Hospital
        add   Director char(30);
```

Observemos que esta operación puede realizarse aun cuando la relación Hospital ya contenga tuplas; en ese caso las tuplas preexistentes tendrán automáticamente asignado el valor nulo en el campo "Director" hasta tanto sean modificadas explícitamente. ¿Qué ocurre si el alter define un atributo restringido a valores no nulos? Convenientemente, está prohibido usar alter para agregar un atributo restringido a valores no nulos.

4.1.2. Consulta

El verbo select, que prefija todas las operaciones de consulta, es el más importante del lenguaje. La forma general de una operación select es la siguiente:

```
select [distinct] ítem(s)
from relación(es)
[where fórmula]
[group by atributo(s) [having fórmula]]
[order by atributo(s)]
```

donde los corchetes indican construcciones opcionales. A pesar de su nombre, select no corresponde a la operación de selección del álgebra relacional, sino que es mucho más poderoso.

En primer lugar, podemos usar el select como simple operador de proyección:

EJEMPLO 4.3:

```
select cargo from Personal;
```

Cargo
Enfermera
Enfermera
Ordenanza
Residente
Enfermera

Notamos que el resultado del select no es, estrictamente hablando, una relación, puesto que algunas tuplas aparecen más de una vez. Como además es posible crear una nueva tabla y asignarle el resultado de un select, se sigue que SQL no implementa realmente relaciones tal como las hemos definido. Esto no representa un problema serio ya que el usuario siempre puede, si lo desea, pedir la eliminación de tuplas duplicadas usando el calificador distinct:

EJEMPLO 4.4:

```
select distinct cargo from Personal;
```

Cargo
Enfermera
Ordenanza
Residente

Si queremos proyectar sobre *todos* los atributos de una relación, por ejemplo, para mostrar el contenido completo de la relación, no necesitamos listar todos los atributos: basta con el símbolo *:

EJEMPLO 4.5:

```
select *
from Medico;
```

La cláusula where sirve para restringir mediante una fórmula el conjunto de tuplas que se desea extraer. Por ejemplo, la consulta siguiente extrae los nombres y números de los pediatras, comportándose como una selección algebraica:

EJEMPLO 4.6:

```
select nombre, numero
from Medico
where spec = "pedia";
```

En la cláusula where pueden utilizarse también condiciones complejas formadas con los operadores and, or y not, y los comparadores =, <, >, ≤, ≥, ≠, >, <.

Los comparadores anteriores se denominan predicados de comparación, entre los que se incluyen también between y su negación not between. El primero de ellos se expresa por la condición

Valor between *Valor1* and *Valor2*

donde *Valor1* = < *Valor2* y la condición será verdadera si *Valor* pertenece al rango [*Valor1*, *Valor2*].

EJEMPLO 4.7: ¿cuáles son los hospitales que tienen entre 150 y 200 camas?

```
select nombre, camas from Hospital
        where camas between 150 and 200;
```

Del mismo modo not between se expresa por:

Valor not between *Valor1* and *Valor2*

donde la condición será verdadera si *Valor* no está en el rango [*Valor1*, *Valor2*].

Otros predicados son null y like, que describiremos brevemente. Null se utiliza para determinar si un componente de una tupla es nulo y su forma es

Nombre de columna is null

Como en la mayoría de los predicados se puede combinar not con null.

El predicado like se utiliza combinado con los caracteres especiales % y _ para la búsqueda de patrones en los datos. Su forma general es la siguiente:

Nombre de columna like "patrón de caracteres"

% y _ son metasímbolos donde % representa cero o más caracteres, mientras que _ representa exactamente un carácter, en el valor de *Nombre de columna*.

EJEMPLO 4.8: supongamos que la tabla Hospital tiene una columna Dirección, en la que se registra el nombre de la calle, el número y la localidad del hospital. Si quisiéramos listar los hospitales de la Av. Córdoba en Capital Federal, podríamos escribir:

```
select nombre
from Hospital
where dirección like "Av. Cordoba%, Capital%";
```

4.2. PREDICADOS DE COMPARACIÓN CUANTIFICADOS

Estos predicados son θ, some, any y all, donde θ es un operador de comparación aritmética, some y any representan al cuantificador existencial y all representa al cuantificador universal. El operador θ acompaña a los cuantificadores mencionados vinculando una expresión con un subselect:

Expresión θ [all | some | any] (Subselect)

El predicado θany o θsome se satisface si existe al menos una fila del resultado del subselect que está en relación θ con *Expresión*. Por otra parte, el predicado θall se satisface si toda tupla del subselect está en relación θ con *Expresión*.

Los siguientes ejemplos clarifican estos conceptos:

EJEMPLO 4.9: listar los nombres de los hospitales que tienen alguna guardia de "Infecciones" o de "Cuidados Intensivos":

```
select nombre
from Hospital
where numero = some (select hosp
                     from Guardia
                     where nombre in [Infecc, CuidInt])
```

EJEMPLO 4.10: listar los nombres de los hospitales que tienen la menor cantidad de camas:

```
select nombre from Hospital
where camas ≤ all (select camas from Hospital);
```

A título de observación agregamos que some ha sido propuesto para reemplazar a any en el estándar SQL92, con la finalidad de ajustar el nombre con el significado del predicado.

4.3. La Operación de Junta en SQL

Para realizar el equivalente a una junta, simplemente listamos más de una relación en la cláusula from e indicamos en la cláusula where las condiciones de la junta. Por ejemplo, para listar todos los pediatras del hospital Central:

Ejemplo 4.11:

```
select Medico.nombre
from Medico, Hospital
where espec = "pedia"
      and Medico.hosp = Hospital.numero
      and Hospital.nombre = "Central";
```

Observemos el uso de "Medico.nombre" para desambiguar el atributo "nombre"; esto es sólo necesario cuando dos relaciones que participan en la consulta tienen un atributo con el mismo nombre, como ocurre en este ejemplo.

Supongamos que tenemos una consulta de las que requieren una junta de una relación consigo misma; por ejemplo, hallar todos los pares de médicos que trabajan en un mismo hospital. Para esto, la relación Médico debe aparecer dos veces en la cláusula from, acompañada de dos identificadores distintos que usaremos para referirnos a las dos "copias" de la relación:

Ejemplo 4.12:

```
select D1.nombre, D2.nombre
from Medico D1, Medico D2
where D1.hosp = D2.hosp;
```


De lo que hemos visto hasta el momento, podemos inferir que la semántica de una operación select es la siguiente. Primero formamos el producto cartesiano de todas las relaciones que aparecen en la cláusula from; si una relación aparece más de una vez, participará más de una vez en el producto cartesiano. Luego seleccionamos de este producto cartesiano aquellas tuplas que satisfacen todas las condiciones indicadas en la fórmula de la cláusula where; y finalmente proyectamos sobre las columnas que aparecen en el encabezamiento del select. Esto nos da una forma inmediata de transformar cualquier consulta select (al menos de las que hemos visto hasta ahora) en una expresión del álgebra relacional. Por ejemplo, la consulta del último ejemplo se transforma en

$$\pi_{3,7}\,(\sigma_{1=5}\,(\text{Médico} \times \text{Médico}))$$

4.3.1 Consultas Anidadas

Existen dos formas más de expresar consultas de junta. Consideremos por ejemplo la consulta que requiere hallar todos los hospitales donde trabaja algún pediatra. De acuerdo con lo presentado hasta ahora, la expresaríamos como:

Ejemplo 4.13:

```
select Hospital.nombre
from Medico, Hospital
where espec = 'ped'
      and Medico.hosp = Hospital.numero
```

Otra posibilidad es usar lo que se llama un select anidado, o sea, un término de la forma

```
atributo in <Sentencia select>
```

como una de las condiciones en la cláusula where:

```
select nombre
from Hospital
where numero in
        (select hosp
        from Medico
        where espec = "pedia");
```

La semántica del in es simplemente pertenencia a un conjunto; la <sentencia select> debe ser tal que produzca una relación unaria, o sea, un conjunto, y la condición

atributo in <sentencia select>

es cierta para una tupla dada si el valor del atributo en esa tupla pertenece al conjunto especificado por la <sentencia select>. En el ejemplo, esto equivale a pedir el conjunto de todos los nombres de hospital tales que su número de hospital pertenece al conjunto de números de hospital que aparecen en la relación Médico con especialidad pediatra. También es admisible la negación del in, de la forma

atributo not in < sentencia select>

La tercera forma de especificar esta consulta es usando el cuantificador existencial exists:

EJEMPLO 4.14:

```
select nombre
from Hospital
where exists
        (select *
        from Medico
        where espec = "pedia"
            and Medico.hosp = Hospital.numero );
```

O sea, listar los nombres de todos los hospitales *h* tales que existe una tupla en la relación Médico con especialidad igual a pediatría y código de hospital igual a *h*. En general, la condición

exists (<sentencia select >)

es cierta exactamente cuando el resultado de la <sentencia select> no es el conjunto vacío de tuplas. La condición exists también puede ser negada en la forma

not exists (<sentencia select>).

Observemos con cuidado la diferencia entre el significado de las variables Médico.hosp y Hospital.número en la última línea del ejemplo anterior: como Médico es una de las relaciones especificadas en la cláusula from del select anidado, este select va a examinar todas las tuplas de la relación Médico buscando las que cumplan la condición. En cambio, como Hospital no aparece en la cláusula from interna, el valor de Hospital.número es constante para cada ejecución del select anidado. En general, el problema de rango de definición de variables suele causar errores en el uso de sentencias anidadas en SQL.

4.3.2. Otros Tipos de Junta

Una de las características de SQL es que suele haber muchas formas distintas de expresar la misma consulta. Por ejemplo, además de los ya mencionados, existen otros tipos de operaciones de junta. Entre ellos se cuentan

1) Natural Join: corresponde a la definición algebraica de esta operación, es decir, cuando los atributos sobre los que se desea realizar la junta tienen los mismos nombres.

EJEMPLO 4.15: listar nombre y cargo del personal que trabaja en guardias de maternidad, en cualquier hospital:

```
select p.nombre, p.cargo
from Personal p natural join Guardia g
where g.nombre = "Mat";
```

Obsérvese que no es necesario incluir las condiciones de junta como en los casos anteriores.

ii) Join con condición: es similar a la junta original ya que puede realizarse sobre cualquier columna, independientemente de los nombres, pero la condición se incluye en una cláusula ON.

EJEMPLO 4.16: cuáles son los nombres de los pediatras y los hospitales (nombres) en los que trabajan:

```
select m.nombre, h.nombre
from Medico m join Hospital h
    on m.hosp = h.numero
where espec = "pedia";
```

iii) Join por nombre de columna: dos tablas pueden tener varias columnas con los mismos nombres. Si se desea realizar la junta usando sólo ciertas columnas, se puede usar este tipo de Join que emplea la cláusula USING para declarar las columnas seleccionadas.

EJEMPLO 4.17: si cambiamos el nombre de la columna *guardia* por *número* en la tabla Personal, ahora Guardia y Personal tienen tres columnas con el mismo nombre: *hosp, número* y *nombre*. Sin embargo, *nombre* representa cosas distintas en cada tabla. Luego, si se desea saber el nombre del personal asignado a cada una de las guardias se podría escribir:

```
select p.nombre, g.nombre
from Personal p join Guardia g
        using (hosp, numero);
```

4.3.3. Outer Join

Todos los operadores de junta mencionados en la sección anterior, pertenecen a la clase de juntas internas o inner joins. En contraposición, existen también las operaciones de outer join. Éstas preservan en el resultado las filas de una tabla que no tienen su correspondiente fila juntable en la otra tabla, dependiendo del tipo particular de operación, a saber:

i) Left Outer Join: preserva las filas correspondientes a la tabla de la izquierda que no tienen su correspondiente fila juntable en la tabla de la derecha.

EJEMPLO 4.18: sean las tablas R y S, tales que

R: A	B
a1	b1
a2	b2

S: C	D
b1	d1
b3	d2

```
select *  from R left outer join S
          on R.B = S.C;
```

el resultado tendrá columnas A B C D y contendrá las filas de la junta de R con S más las filas no juntables de R completadas con null:

Resultado:

A	B	C	D
a1	b1	b1	d1
a2	b2	null	null

ii) Right Outer Join: realiza la junta y además preserva las filas no juntables correspondientes a la tabla derecha.

EJEMPLO 4.19:

```
select *  from R right outer join S
          on R.B = S.C;
```

En este caso el resultado es

A	B	C	D
a1	b1	b1	d1
null	null	b3	d2

iii) Full Outer Join: se comporta como una combinación de las dos operaciones anteriores.

EJEMPLO 4.20:

```
select *   from R full outer join S
        On R.B = S.C;
```

Resultado:

A	B	C	D
a1	b1	b1	d1
a2	b2	null	null
null	null	b3	d2

Si utilizamos como ejemplo las tablas Hospital y Médico, podríamos requerir los nombres de los médicos y los hospitales en que trabajan.

EJEMPLO 4.21: si algún médico no trabaja en ningún hospital, su nombre debería aparecer de todos modos, indicando esa situación. La consulta SQL correspondiente es la siguiente:

```
select m.nombre, h.nombre
from Medico m left outer join Hospital h
        on m.hosp = h.numero;
```

EJEMPLO 4.22: si algún hospital no tiene todavía ningún médico, tal vez porque esa información no fue aún ingresada, deseamos que en el resultado aparezcan también esos hospitales, aunque no los médicos no asignados a un hospital:

```
select m.nombre, h.nombre
from Medico m right outer join Hospital h
  on m.hosp = h.numero;
```

EJEMPLO 4.23: combinando los dos ejemplos anteriores tenemos una consulta donde pedimos los médicos y los hospitales en que trabajan, pero también tanto los médicos que no trabajan en ningún hospital como los hospitales que no tienen ningún médico:

```
select m.nombre, h.nombre
from Medico m full outer join Hospital h
  on m.hosp = h.numero;
```


4.4. OTROS OPERADORES

Analizaremos brevemente los operadores de union, union join, intersect y except.

i) Union: permite combinar dos relaciones o tablas cuyas columnas tienen tipos de datos comparables. Las tablas a considerar pueden ser las tablas de origen o las que resultan de la ejecución de las subconsultas participantes. Realiza la unión matemática de las tuplas que son el resultado corriente de las correspondientes subconsultas.

EJEMPLO 4.24: sean R y S las tablas

R:

A	B
a1	b1
a2	b2

S:

A	B
a2	b2
a3	b3

```
select *
from R
      union
select *
from S;
```

El resultado está dado por

A	B
a1	b1
a2	b2
a3	b3

Se puede observar que este operador elimina filas duplicadas, aunque éstas se podrían conservar usando la cláusula all a continuación de union.
Existe también una cláusula corresponding que permite unir dos tablas que no tienen exactamente el mismo esquema. Seguida por una lista de

nombres de columnas, incluirá en la unión sólo las columnas indicadas, exactamente como si se realizara previamente una proyección. Por el contrario, corresponding *sin* ninguna lista de nombres de columnas, producirá una tabla con solamente las columnas comunes a ambas tablas. Su forma general es:

```
select *
from R
  union corresponding [by (columnai1, columnai2, ...
                                        columnaik)]
select *
from S;
```

ii) Union Join: toma dos tablas R y S, y produce una tercera cuyas columnas resultan de la unión de las columnas de R y S. Cada fila de la nueva tabla proviene de la unión de las filas de R y de S extendidas con null.

EJEMPLO 4.25: sean R y S como en el ejemplo 4.18

```
select * from R union join S
```

Resultado:

A	B	C	D
a1	b1	null	null
a2	b2	null	null
null	null	b1	d1
null	null	b3	d2

iii) Intersect: dadas dos tablas R y S da como resultado las tuplas que están en la intersección de ambas:

```
select * from R
 intersect
select * from S;
```

También aquí, como en el caso del operador union, es posible combinar intersect con all y corresponding.

iv) Except: permite calcular la diferencia entre dos tablas R y S:

```
select * from R
      except
select * from S;
```

Si bien esta operación es posible realizarla fácilmente utilizando el predicado not exists, en la definición del estándar SQL92 se prefirió incluir un operador explícito. Al igual que en el operador anterior, se puede combinar except con all y corresponding.

4.5. UNA FORMA DE IMPLEMENTAR EL OPERADOR DE DIVISIÓN

Hasta el momento hemos visto cómo es posible expresar en SQL todas las operaciones del Álgebra Relacional, excepto la *división*. Esta operación incluye, en términos del cálculo de predicados, un cuantificador universal del que SQL no dispone (no confundir con el predicado cuantificado all), lo que complica bastante su representación. Por otra parte SQL sí dispone, como se vio, de una construcción equivalente al cuantificador existencial: el predicado exists. Recordemos que exists verifica si el subselect que le sigue es vacío o tiene alguna fila como resultado. En el primer caso el predicado resulta falso y en el segundo verdadero.

Al tratar el operador de división en el Álgebra Relacional vimos un ejemplo en el que pedíamos los nombres de los médicos que trabajaban en todos los hospitales. Esa misma consulta fue planteada también en cálculo relacional (de dominios). Nos remitiremos a ese ejemplo, pero utilizaremos el Cálculo Relacional de Tuplas, es decir, las variables ahora representarán tuplas, en lugar de dominios. Luego la consulta en CRT resulta:

$$\{ m.nombre \mid Médico(m) \ \& \ (\forall h)(Hospital(h) \rightarrow (\exists n)(Médico(n) \ \& \ n.hosp = h.código \ \& \ m.nombre = n.nombre)) \}$$

lo que resulta lógicamente equivalente a

$$\{ m.nombre \mid Médico(m) \ \& \ (\forall h)(\neg Hospital(h) \vee (\exists n)(Médico(n) \ \& \ n.hosp = h.código \ \& \ m.nombre = n.nombre)) \}$$

lo que a su vez resulta en

$$\{m.nombre \mid Médico(m) \ \& \ \neg(\exists h)(Hospital(h) \wedge \neg(\exists n)(Médico(n) \ \& \ n.hosp = h.código \ \& \ m.nombre = n.nombre))\}$$

Luego en SQL

EJEMPLO 4.26:

```
select nombre
from Medico e
where not exists
        (select *
        from Hospital o
        where not exists
                (select *
                from Medico e1
                where e.nombre = e1.nombre
                    and o.codigo = e1.hosp));
```

Veamos otro ejemplo:

EJEMPLO 4.27: supongamos que tenemos tres tablas, correspondientes a los esquemas Estudiante (Nro, Nombre, ...), Materia (Código, Nombre, Año, ...) y Cursa (Nro, Código) y deseamos saber los nombres de los alumnos que cursan *todas* las materias de 4° año.

En Álgebra Relacional, la respuesta se obtendría evaluando la expresión

$$\pi_{Nombre}(Estudiante \mid\times\mid (Cursa \mid \sigma_{Año=\text{"4°"}}(Materia)))$$

Veamos cómo se expresa ahora en SQL:

NO el alumno ok matrícula está dentro de cursa


```
select nombre
from Estudiante e
where not exists
        (select *
        from Materia m
        where m.Año = "4°" and not exists
                (select *
                from Cursa c
                where c.Nro = e.Nro and
                    c.Codigo = m.Codigo));
```

4.6. USO DE FUNCIONES

Hasta ahora, todas las consultas que hemos visto se pueden expresar sin problema en álgebra relacional. SQL provee además funciones que permiten aplicar a una relación unaria un operador tal como suma, cuenta, promedio, máximo, mínimo, obteniendo como resultado un número. Por ejemplo, si queremos saber cuántos hospitales hay registrados en la base de datos:

EJEMPLO 4.28:

```
select count(*)
from Hospital;
```

Esto significa: seleccionar todas las tuplas de la relación Hospital y mostrar el resultado de aplicar al conjunto de tuplas resultante el operador count, que simplemente las cuenta. Notemos que si hay tuplas duplicadas, serán contadas tantas veces como aparezcan.

El operador count puede aplicarse a un conjunto de tuplas de cualquier rango: en cambio, los operadores aritméticos avg, sum, max, min se aplican solamente a conjuntos de tuplas unarias. Por ejemplo, ¿cuál es el número promedio de camas tomado sobre todos los hospitales?

EJEMPLO 4.29:

```
select avg (camas)
from Hospital;
```

¿Qué hospitales tienen más camas que el valor promedio?

EJEMPLO 4.30:

```
select nombre
from Hospital
where camas >
        (select avg (camas)
         from Hospital);
```

Supongamos que tenemos también una relación "Paciente", uno de cuyos atributos es la fecha de nacimiento "nacido". Usando el operador max, podemos averiguar la edad del paciente o pacientes más jóvenes:

EJEMPLO 4.31:

```
select max (nacido)
from Paciente;
```

Si en cambio queremos saber quiénes son los pacientes más jóvenes, usamos max dentro de un select anidado:

EJEMPLO 4.32:

```
select nombre
from Paciente
where nacido =
        (select max (nacido)
         from Paciente);
```


4.7. ORDEN Y AGRUPAMIENTO

Uno de los requerimientos prácticos más simples es el de examinar el resultado de una consulta en cierto orden: alfabético por nombre o en orden descendente de fecha, o lo que fuera. Ésta es la función de la cláusula order by, que no discutiremos en detalle.

Menos obvia es la necesidad de agrupar conjuntos de tuplas que tengan cierto valor en común y aplicar luego una función de conjunto, tal como count, separadamente a cada conjunto. Por ejemplo, supongamos que queremos saber el número de pediatras en cada hospital:

EJEMPLO 4.33:

```
select hosp, count (numero)
from Medico
where espec = "pedia"
group by hosp;
```

Conceptualmente, la semántica del group by es que la relación se divide en grupos disjuntos de acuerdo con el valor del atributo hosp; la consulta luego se aplica separadamente a cada uno de los grupos. La lista de ítems que sigue al verbo select debe tener la propiedad de que cada uno de los ítems genera un único valor para cada grupo; por ejemplo, en este caso, hemos agrupado por número de hospital, por lo que cada grupo tiene un único valor de "hosp" y un único valor de "count (número)".

Del mismo modo que la cláusula where se aplica en un bloque select-from como un filtro, la cláusula having cumple ese papel para la tabla agrupada. Su formato es

having *condición_de_búsqueda*

donde *condición_de_búsqueda* se aplica a cada grupo producido por group by

Ejemplo 4.34: listar los hospitales que tienen más de tres psiquiatras, incluyendo la cantidad de especialistas de que disponen:

```
select hosp, count (numero)
from Medico
where espec = "psiqu"
group by hosp having count (numero) > 3;
```

Obsérvese que la cláusula having tiene efecto a posteriori de la acción del group by. Luego having actúa sobre los grupos obtenidos, incluso no existiendo el group by, en cuyo caso se considera un único grupo.

4.8. ACTUALIZACIÓN

SQL provee tres sentencias que permiten modificar el contenido de una relación o relaciones: insert, delete y update, cuya sintaxis se resume como sigue:

```
insert into relación [campo(s)] values (constante(s))
insert into relación [campo(s)] sub-consulta;

delete from relación [where condición];
update relación set [campo = exp] [where condición];
```

Supongamos que queremos simplemente agregar una nueva tupla a la relación Hospital, cuyos valores conocemos:

EJEMPLO 4.35:

```
insert
into Hospital
values (29,Clinicas,640);
```

Tenemos también la posibilidad de insertar en una relación, preexistente o no, el resultado de una consulta select:

EJEMPLO 4.36:

```
insert
into Hospital (numero)
    select numero
    from Medico;
```

El efecto de este insert será que, por cada tupla *t* de la relación Médico, se agregará a la relación Hospital una tupla con el mismo número de hospital que *t*, y con valores nulos en los otros atributos.

El update se utiliza para modificar tuplas existentes. Puede afectar a una o múltiples tuplas; por ejemplo, si queremos modificar la cantidad de camas del hospital número 29:

EJEMPLO 4.37:

```
update Hospital
set    camas = 299
where  numero = 29;
```

Si la cláusula where admite más de una tupla, todas aquellas que satisfagan la condición serán actualizadas. Por ejemplo, si tenemos una relación "Empleado" y queremos aumentar el sueldo en un diez por ciento a todos los analistas:

EJEMPLO 4.38:

```
update Empleado
set    sueldo = 1.1 *sueldo
where  cargo = "analista";
```

Finalmente, la función delete elimina tuplas de una relación. Así como update, puede llevar asociada una condición where que afecte a una o a múltiples tuplas. Por ejemplo, la siguiente sentencia elimina todos los psiquiatras:

EJEMPLO 4.39:

```
delete
from Medico
where espec = "psiqu";
```

4.9. DEFINICIÓN DE VISTAS

Una base de datos tiene grupos de usuarios diversos, con requerimientos y privilegios también diversos. Es muy conveniente poder construir para un grupo de usuarios una *vista* de la base de datos, o sea, una relación virtual definida por la aplicación de una consulta a las relaciones reales o *relaciones de base*. Una vista es una forma de mirar la base de datos que resulta por alguna razón más natural o conveniente que el diseño original. Combinando la posibilidad de definir vistas con la de controlar qué usuarios tienen acceso a qué vistas, que veremos en la sección próxima, obtenemos un mecanismo poderoso de adecuación de una misma base de datos a distintos requerimientos y distintos niveles de privilegio de sus usuarios.

La vista más simple es aquella que se define, en términos algebraicos, por la aplicación de una selección y una proyección a una única relación de base:

EJEMPLO 4.40:

```
create view Pediatra
as      select nombre, numero, hosp
        from Medico
        where espec = "pedia";
```

El efecto de esta sentencia es la creación de una relación virtual llamada "Pediatra" con tres atributos llamados "nombre", "número", y "hosp", cuyos tipos de datos son exactamente los mismos que los de los correspondientes atributos de la relación "Médico". Esta nueva relación contiene todas las tuplas de la relación "Médico" cuya especialidad sea pediatría. Esta correspondencia entre "Médico" y "Pediatra" es dinámica: si insertamos una nueva tupla correspondiente a un pediatra en "Médico", esta tupla se materializa "mágicamente" también en "Pediatra", y si damos una de baja de "Médico", también desaparece de "Pediatra".

Una vez definida la vista, a los ojos del usuario ésta se comporta exactamente como cualquier relación de base en cuanto a las consultas; por ejemplo la consulta siguiente cuenta el número de pediatras:

EJEMPLO 4.41:

```
select count(*)
from Pediatra;
```

Una vista puede ser definida sobre más de una relación; por ejemplo, supongamos que queremos, por conveniencia, tener una tabla de pares de nombres de hospital con nombres de médicos que trabajan en ese hospital:

EJEMPLO 4.42:

```
create view Hosp_Med(hosp,med)
as      select Hospital.nombre, Medico.nombre
        from Hospital, Medico
        where hospital.numero = Medico.hosp;
```

Notemos que en este caso hemos debido asignar nuevos nombres a los atributos de la vista usando la construcción "Hosp_Med(hosp,med)" para evitar la confusión de dos atributos llamados "nombre".

Hemos visto que, a los fines de consulta, una vista es indistinguible de una relación de base. ¿Qué ocurre si queremos actualizar una vista? Consideremos primero la vista "Pediatra". La sentencia de actualización:

EJEMPLO 4.43:

```
insert
into Pediatra
values ("38", "607", "Bochini");
```

tiene una interpretación unívoca en términos de actualización de la relación de base "Médico": basta con insertar en ésta una nueva tupla con número de hospital 38, número de médico 607, nombre Bochini, y especialidad pediatría para obtener el efecto deseado. Lo mismo ocurre para sentencias dele-

te y update. Consideremos en cambio la vista "Hosp_Med", y la siguiente actualización:

EJEMPLO 4.44:

```
delete
from      Hosp_Med
where     med = "Artime"
      and hosp = "Italiano";
```

O sea, queremos eliminar la asociación entre el doctor Artime y el Hospital Italiano. ¿Qué modificaciones podemos hacer a las relaciones de base de modo de eliminar la tupla <Artime, Italiano> de la vista?

Podríamos simplemente eliminar la tupla de Artime de la relación Médico; o, más drásticamente, eliminar la tupla del Hospital Italiano de la relación de hospitales; o asignar a Artime a algún otro hospital, o quizás cambiar el número de hospital del Hospital Italiano. En general, no hay forma en que el sistema pueda decidir cuál es la interpretación más razonable de una actualización de este tipo. Las implementaciones de SQL y otros lenguajes relacionales generalmente evitan el problema restringiendo el tipo de vistas sobre las cuales se pueden hacer actualizaciones a aquellas definidas sobre una sola relación. Desde un punto de vista teórico, el problema de cuáles vistas pueden ser actualizadas en forma unívoca no está aún completamente resuelto.

4.10. CONTROL DE SEGURIDAD

SQL provee dos sentencias, grant y revoke, que permiten controlar qué tipo de acceso se permite a qué usuario sobre relaciones, sean de base o vistas. Para poder ejecutar cualquier operación de SQL, el usuario debe poseer la autoridad necesaria. Por ejemplo, para ejecutar la consulta

EJEMPLO 4.45:

```
select *
from Medico;
```


el usuario debe tener autoridad select sobre la relación Médico. Para actualizar, se requieren las autoridades insert, delete o update; existe también autoridad para crear, alterar o eliminar una relación, y muchos otros tipos que no discutiremos. Para otorgar lal autoridad select sobre la relación Médico al usuario Pepe, el creador de la relación debe ejecutar la siguiente sentencia:

EJEMPLO 4.46:

```
grant select on table Medico
      to Pepe;
```

que puede ser revocada más tarde de la siguiente manera:

EJEMPLO 4.47:

```
revoke select on table Medico
      to Pepe;
```

Una opción interesante es la de otorgar a un usuario una cierta autoridad y además el privilegio de pasar esa autoridad a otros usuarios:

EJEMPLO 4.48:

```
grant select, update on table Hospital, Medico to Pepe, Luisa
with grant option;
```

Esto otorga a Pepe y Luisa, no sólo la autoridad de consultar y modificar las tablas mencionadas, sino también la de poder a su vez pasar estos privilegios a otros usuarios.

4.11. VALORES NULOS

Al ingresar nuevas tuplas en las relaciones, podría ocurrir que el dato correspondiente a una columna o atributo fuera *desconocido* o todavía *no definido* en ese momento, o *inaplicable*. Como ejemplo, supongamos que se desea incorporar un nuevo hospital en la relación Hospital, pero en ese momento no sabe-

mos cuántas camas tiene, o se trata de un hospital que atiende exclusivamente pacientes ambulatorios, por lo que no dispone de camas. Para poder ingresar las tuplas de todas maneras, éstas se pueden completar con un valor especial denominado valor nulo, representado por el término null.

SQL provee una facilidad uniforme para el manejo de valores nulos. Por otra parte, un sgbd representa todo valor nulo con la misma representación interna independientemente de la base de datos, relación o atributo de que se trate.

Analizaremos ahora como se comportan las distintas operaciones en SQL ante la presencia de valores nulos.

Supongamos que tenemos la siguiente consulta:

```
select nombre from Hospital where camas < 100 or camas >=100;
```

Ante esta sentencia, el sgbd debería listar todos los hospitales que tienen consignada una cantidad de camas. ¿Qué ocurre con los hospitales que tienen null en el atributo camas? Para esos hospitales, el predicado de comparación evaluará a un valor especial denominado unknown (desconocido). La única forma de listar esos hospitales es utilizando el predicado is null que, en este caso, permite determinar si el atributo camas contiene el valor null:

```
select nombre from Hospital where camas is null;
```

Las comparaciones en la presencia de nulos devuelven uno de tres valores posibles: *true, false y unknown* (verdadero, falso y desconocido). Así, por ejemplo, si se compara un valor cualquiera con un nulo, como ya se vio, el resultado es *unknown* y el mismo resultado se obtiene de la comparación de dos valores nulos. Si un predicado consta de dos comparaciones unidas por or y una resulta *true* y la otra *unknown*, el predicado completo evaluará a *true*. Si las dos comparaciones estuvieran unidas por and y una resulta *false* y la otra *unknown*, el predicado evaluará a *false*.

En la tabla de la figura 4.1 damos la definición de los operadores lógicos en este sistema trivaluado.

En general, al tratar con valores nulos debemos tener en cuenta que en cualquier operación aritmética donde uno de sus operandos es nulo el resultado será *null*; cualquier comparación en la que participe un valor nulo tendrá como valor de verdad a *unknown*; las operaciones lógicas en las que participe *unknown* se rigen por la tabla 4.1; las cláusulas where y having habilitan tuplas o grupos de tuplas cuando evalúan exactamente a verdadero.

Analizaremos brevemente cómo se comportan las funciones agregativas de SQL en la presencia de valores nulos. Las funciones **min**, **max**, **sum** y **avg** ignoran los valores nulos, a menos que todos los valores de la columna involucrada sean nulos. En este último caso, el resultado es nulo. La función **count**, a diferencia de las otras, cuenta la cantidad de filas independientemente de su contenido; pero si la columna a la que se aplica **count** sólo contiene nulos, el resultado es 0 (cero).

p	q	¬p	p∧q	p∨q
V	V	F	V	V
V	F	F	F	F
V	D	F	D	V
F	V	V	F	V
F	F	V	F	F
F	D	V	F	D
D	V	D	D	V
D	F	D	F	D
D	D	D	D	D

Figura 4.1
Tabla de verdad para una lógica de tres valores
(V: verdadero F: falso D: desconocido)

En la siguiente sección, en la que trataremos el tema de *restricciones*, analizaremos cómo excluir a null como valor corriente de un atributo.

4.12. RESTRICCIONES

Una de las consideraciones más importantes a tener en cuenta con respecto a una base de datos es la de la integridad de los datos que contiene. Existen al menos dos formas de mantener la integridad de una bd. Una de ellas es sumergir las restricciones en los programas de aplicación que explo-

tan la bd. En este caso será necesario multiplicar el código específico que fuerza las restricciones en todas las aplicaciones que acceden a esa bd.

La otra forma es mediante declaraciones al sgbd. Esas declaraciones, muchas de ellas incluidas en el create de las tablas o relaciones, forman parte de los metadatos de la bd, y se almacenan en el respectivo catálogo como predicados SQL.

Analizaremos brevemente los distintos tipos de restricciones previstos en SQL.

i restricciones aplicables a atributos

- not null especifica que no es posible ingresar un valor nulo para el atributo.

  EJEMPLO 4.49:

  create table Hospital (numero smallint not null, ...)

  Por esta declaración no se admitirá un valor nulo como número de hospital.

- Unique especifica que los valores del atributo son únicos. Luego, no puede haber dos o más filas con el mismo valor para ese atributo.

  EJEMPLO 4.50:

  create table Hospital (numero smallint unique, ...)

  En este caso se asegura que el número de hospital sea único, pero puede haber uno con null. Para que tampoco pueda haber un valor nulo en la columna de número, especificamos

  create table Hospital (numero smallint unique not null, ...)

- Check permite especificar diferentes tipos de reglas de integridad referidas a un atributo. La sintaxis correspondiente es

  check (*condición_de_búsqueda*),

donde la condición de búsqueda puede estar integrada por diversas expresiones.

EJEMPLO 4.51:

Create table hospital (..., camas smallint check (camas < 1000), ...)

El sgbd no admitirá ninguna tupla que haga falsa la condición. Pero sí lo hará si la evaluación resulta verdadera o desconocida (true o unknown).
También es posible construir la condición de búsqueda con consultas que impliquen funciones agregativas sobre otras tablas.

EJEMPLO 4.52:

```
create table hospital (....., camas smallint
        check (camas <= (select count(*)/ 10
                         from Personal)), ....)
```

Esta restricción limita la cantidad de camas al 10% de la cantidad de personal.
Supongamos que se desea crear una tabla Pacientes para una empresa de medicina prepaga:

EJEMPLO 4.53:

```
create table Paciente (....., cuota decimal (7,2), desc_farmacia smallint,
        consultas_sin_cargo smallint, ....,
    check ((cuota between 50 and 500) and
        (desc_farmacia between 10 and 100) and
        (consultas_sin_cargo between 5 and 20))
```

Observamos en este ejemplo que existen varias condiciones y, a su vez, cada una utiliza un rango de valores: la primera limita la cuota a valores entre 50 y 500; la segunda limita los descuentos en farmacia a porcentajes entre 10 y 100; la última expresa que las consultas sin cargo pueden estar entre 5 y 20.
También es posible especificar en la condición de búsqueda

una lista de valores, como en el siguiente ejemplo:

EJEMPLO 4.54:

```
create table Medico (..., espec char(5)
        check (espec in ('clini', 'pedia', infec',
                         'obste', 'cardi', 'otras'), ...),
```

ii. restricciones de dominio: es posible crear dominios específicos a una aplicación limitando los tipos nativos del sgbd por medio de restricciones. Luego se usan esos dominios para especificar los valores permitidos para los atributos en la definición de las tablas.

EJEMPLO 4.55:

```
create domain clave as integer check (value > 0) check (value
                                                  is not null)
create domain color as char(10) default 'SinColor'
        check (value in ('Sincolor', 'Azul', 'Gris', 'Rojo', 'Verde'))
```

iii. aserciones: permiten definir restricciones aplicables a varias tablas. Este tipo de restricción, denominada assertion, se crea por medio de una sentencia create con la siguiente sintaxis:

```
create assertion nombre_de_restricción
        check (condición_de_búsqueda)
```

EJEMPLO 4.56: supongamos que en las tablas Personal y Médico tenemos un atributo Sueldo. La restricción es que la suma de los sueldos del personal médico y no médico no debería superar el presupuesto de 10.000.000 pesos. Para esto podemos crear una aserción:

```
create assertion total_sueldos
        check ((select sum (sueldo) from Personal) + (select sum
                                                      (sueldo)
                                  from Medico)  <= 10 000 000)
```


iv. clave primaria: permite declarar un atributo o conjunto de atributos como clave primaria de una tabla. Su uso consiste simplemente en agregar primary key a la definición del atributo o declarar primary key a una lista de atributos.

EJEMPLO 4.57:

```
create table Hospital (Numero smallint primary key, ...)
create table Personal (Hosp smallint, Nro smallint,
                       nombre varchar(40), ...,
                       primary key (Hosp, Nro), ...)
```

v. integridad referencial: para analizar el concepto de integridad referencial, debemos previamente revisar la noción de clave foránea o clave externa. Una clave foránea se define por uno o más atributos que permiten vincular una tabla a otra, en la que esos atributos constituyen una clave. Por ejemplo, en nuestras tablas Médico y Personal, el atributo Hosp que representa el código de hospital es una clave foránea. Las claves foráneas permiten implementar relaciones uno a muchos. Veamos cómo expresar esto en SQL:

EJEMPLO 4.58:

```
create table Médico (Hosp smallint, Nro smallint,
                     Nombre varchar(40), ...,
                     constraint cf_hosp foreign key (hosp)
                     references Hospital (número), ...)
```

Obsérvese que hemos declarado a cf_hosp como el nombre de la restricción.

Nombres de restricciones

Es posible asignar un nombre a cada una de las restricciones planteadas anteriormente. Más aún, se considera muy conveniente hacerlo, ya que el

sgbd asigna nombres arbitrarios a las restricciones que no lo tienen. Estos nombres son luego utilizados por el sgbd para informar acerca de violaciones a las restricciones: si el nombre ha sido asignado por nosotros, los mensajes de error serán más significativos.

EJEMPLO 4.59:

```
create table hospital (...., camas smallint
constraint maxcamas check (camas < 1000), ....)
```

otro

```
create table Medico (...., espec char(5) constraint espec_válida
    check (espec in ('clini', 'pedia', infec', 'obste', 'cardi', 'otras'), ....);
```

Acciones referenciales

Supongamos que a cada paciente de nuestra tabla *Paciente* le corresponde un hospital principal al cual debe acudir para su atención. Ese hospital es asignado al momento de la incorporación del paciente al sistema. Luego, si por algún motivo ese hospital se da de baja, se produciría un problema con la restricción de integridad referencial. Veamos como figuraría ésta en la creación de la tabla *Paciente*:

```
create table Paciente (Numero integer not null,
                       Nombre char(30) not null,
                       hosp smallint,

constraint cf_hosp foreign key (hosp)
references hospital (numero), ....)
```

En este caso no es posible eliminar un hospital, en la medida que tenga algún paciente asignado.

Para resolver este tipo de problema, SQL-92 ha previsto las llamadas *acciones referenciales*. Éstas permiten actualizar en forma automática las tablas, ante una eliminación o una actualización, de forma tal de mantener la integridad referencial.

El formato completo para una restricción de integridad referencial es

```
foreign key (c1, ...., ck)
references R (d1, ...., dk)
[on delete [set default | set null | cascade | no action]]
[on update [set default | set null | cascade | no action]]
```

Si declaramos on delete y/o on update, cuando se elimine o actualice una fila en la tabla referenciada R, el sgbd ejecutará la acción especificada. Las acciones alternativas son: set default, set null, cascade y no action. Observemos que no action es la acción por defecto en el caso de no declarar nada. A continuación describiremos brevemente las acciones enunciadas:

i. set default / set null (reemplazar por un valor prefijado/reemplazar por un valor nulo): al eliminar o modificar una fila referenciada en una restricción de integridad referencial, el correspondiente valor en las filas referenciantes es reemplazado por un valor prefijado o valor por defecto si la acción es set default, o por null si la acción especificada es set null.

EJEMPLO 4.60:

```
create table Paciente (.... hosp smallint default 21
                           references Hospital
                           on delete set default, ....)
```

Al dar de baja un hospital, las tuplas de los pacientes asignados a ese hospital se modificarán de manera que el código de hospital resulte 21.

EJEMPLO 4.61:

```
create table Paciente (....
                       hosp smallint
                       references Hospital
                       on delete set null, ....)
```

Si se da de baja un hospital, las tuplas de los pacientes asignados a ese hospital serán modificadas por el sgbd para contener null en el código de hospital, al menos hasta que se los asigne a otro

ii. cascade (aplicar la acción en cascada): al eliminar una tupla referenciada o modificarse su clave, las tuplas referenciantes son eliminadas, en el primer caso, o el correspondiente valor de clave foránea es actualizado acordemente en las tuplas referenciantes, en el segundo

EJEMPLO 4.62: en la tabla Médico, la especialidad se encuentra codificada y existe una tabla de especialidades. Se desea que ante la eliminación de una especialidad, los médicos también sean eliminados de la base de datos.

```
create table Medico (... espec char(5), ...,
                    constraint cf_espec foreign key (espec)
                    references Especialidad on delete
                                                cascade, ...)
```

Por otra parte, si se desea que una modificación del código de una especialidad se traslade a las tuplas de los médicos que tienen esa especialidad deberíamos escribir:

EJEMPLO 4.63:

```
create table Medico (... espec char(5), ...,
                    constraint cf_espec foreign key (espec)
                    references Especialidad on update
                                                cascade ...)
```

4.13. PROGRAMACIÓN SQL

Hasta ahora hemos mostrado el uso de SQL como lenguaje interactivo. Uno de los objetivos del diseño del lenguaje fue que pudiera usarse, básicamente con la misma sintaxis, ya fuera en forma interactiva como desde un programa escrito en algún lenguaje de programación. Sin entrar en detalles,

veamos un ejemplo típico de un programa escrito en un lenguaje de alto nivel similar a Pascal mezclado con sentencias SQL:

EJEMPLO 4.64:

```
exec sql declare M cursor for
        select numero, nombre, camas
        from Hospital
        where camas > 800;

exec sql open M;
        while (M no vacio) do
                exec sql fetch M into :hnum,:hnombre,:hcamas;
                /* procesar esta tupla */
                ...

        end;

exec sql close M;
```

Cada operación SQL comienza con el prefijo exec sql. La variable M es un *cursor*, asociado por la sentencia declare con cierta consulta select. Esta consulta select se ejecuta cuando el programa alcanza la sentencia open para el cursor M. Esto genera un conjunto de tuplas; en este ejemplo, las tuplas de Hospital que tienen más de 800 camas. Para acceder a estas tuplas desde el programa, se usa la sentencia fetch: cada fetch extrae una nueva tupla de este conjunto, asignando sus atributos a las variables :hnum, :hnombre, :hcamas, cuyos nombres comienzan con ":" para indicar que son variables de SQL y no simplemente variables del lenguaje de programación.

El formato de fetch es

fetch [[*orientación*] from] *nombre_de_cursor*
into *variables*

donde *orientación* puede ser next, prior, first o last lo que permite recuperar la próxima la anterior, la primera o la última tupla asociada con el cursor, respectivamente. También aquí puede especificarse absolute o relative y un valor entero indicando en el primer caso la tupla ubicada en la posi-

ción absoluta dada por valor, y en el segundo la tupla ubicada en la posición dada por valor con respecto a la posición actual del cursor.

Para que pueda especificarse una orientación en fetch, el cursor debe ser declarado *"scrollable"*:

Declare *nombre_de_cursor* scroll cursor for ...;

No sólo la sentencia select, sino cualquier sentencia SQL usable en forma interactiva puede usarse también desde un lenguaje de programación. La elección de un lenguaje particular depende por supuesto de la implementación; típicamente las implementaciones de SQL proveen interfaces con los lenguajes COBOL, PL/I, C, FORTRAN, etc.

4.14. SQL DINÁMICO

Este nombre se aplica a la característica que permite crear sentencias SQL en tiempo de ejecución de un programa que accede a una base de datos. En contraposición, las sentencias SQL que son definidas e incluidas completamente en un programa de aplicación que es precompilado, compilado, etc., se denominan SQL *estático*.

En el SQL dinámico se trata, en general, de sentencias que no se conocen, al menos en su totalidad, hasta el momento de ser ejecutadas dentro de una aplicación. Luego deben ser completadas, de alguna manera, o ingresadas totalmente por el usuario, en el preciso instante de su ejecución. Esta característica es sumamente útil en un ambiente de soporte a la toma de decisiones, donde las consultas posibles pueden ser tantas que no resulta práctico intentar considerarlas a todas en el programa de aplicación.

Existen dos enfoques básicos para la ejecución de sentencias en SQL dinámico. En el primero, se prepara la sentencia para ejecución y luego se la ejecuta varias veces. En el segundo, la preparación y la ejecución ocurren en un único paso denominado de *ejecución inmediata*. A continuación analizaremos brevemente cada uno de los enfoques referidos.

En el enfoque de ejecución inmediata se utiliza la sentencia Execute immediate que tiene el siguiente formato:

Execute immediate *nombre_de_variable*

donde *nombre_de_variable* es un parámetro o variable de SQL que contiene un string de caracteres representando la sentencia SQL que se desea ejecutar dinámicamente.

EJEMPLO 4.65: actualizar la cantidad de camas del hospital número 29.

```
Exec SQL Execute immediate 'Update Hospital set camas = 310
                            where numero = 29'
```

Lo que también se puede escribir como

```
var1 = 'Update Hospital set camas = 310 where numero = 29';
Exec SQL Execute immediate :var1;
```

Se utiliza este enfoque si se conoce con anticipación que la sentencia se ejecutará una sola vez. De esta manera se evita desperdiciar recursos del sistema durante toda una sesión, ya que éstos serán liberados inmediatamente después de la ejecución de la sentencia. Por el contrario, si se prevé ejecutar una sentencia un número de veces mayor que uno debería optarse por el otro enfoque, el que se describe a continuación.

En el enfoque de preparación y ejecución múltiple se utilizan las sentencias prepare y execute. El formato simplificado de esas sentencias es

prepare *nombre_de_sentencia* from *nombre_de_variable*

donde *nombre_de_sentencia* es un identificador de la sentencia que será luego usado con el comando execute, mientras *nombre_de_variable* puede ser una variable SQL o un valor dado por un string de caracteres representando la sentencia a ejecutar dinámicamente.

execute *nombre_de_sentencia*

EJEMPLO 4.66: incrementar en 1 la cantidad de camas de ciertos hospitales.

```
Exec SQL prepare sent1
        from 'update Hospital set camas = camas + 1
             where numero = ?'
```

lo que también se puede escribir como

```
var1 = "sent1";
var2 = "update Hospital set camas = camas + 1
                where numero = ?";
Exec SQL prepare :var1 from :var2;
```

Luego para la ejecución escribimos

```
Exec SQL execute sent1 using :cod_hosp;
```

o

```
Exec SQL execute :var1 using :cod_hosp;
```

Observemos que la sentencia SQL contiene un *parámetro dinámico* representado por el signo de interrogación "?". Este parámetro es completado con los sucesivos números de hospitales contenidos cada vez en la variable cod-hosp.

El comando prepare posibilita que la sentencia SQL sea analizada y que el sgbd realice su optimización. El resultado del prepare es una cantidad de información que permite la ejecución posterior de la sentencia. Los recursos utilizados, en este caso, son liberados al final de la sesión

EJERCICIOS

E1 Dado el siguiente esquema de base de datos relacional

Almacén (Nro, Responsable)
Artículo (Cód_art, Descripción, Precio)
Material (Cód_mat, Descripción)
Proveedor (Cód_prov, Nombre, Domicilio, Ciudad))
Tiene (Nro, Coód_art, Cantidad)
Compuesto_por (Cód_art, Cód_mat)
Provisto_por (Cód_mat, Cód_prov)

Realizar las siguientes consultas en SQL:

i) Hallar los nombres de los proveedores de la ciudad de La Plata
ii) Hallar los materiales, código y descripción, provistos por proveedores de la ciudad de Rosario.
iii) Hallar códigos y descripciones de los artículos compuestos por al menos un material provisto por el proveedor López.
iv) Hallar los códigos y nombres de proveedores que proveen al menos un material que se usa en algún artículo cuyo precio es mayor que $100.
v) Hallar el o los códigos de los artículos de mayor precio.
vi) Ídem v) de menor precio.
vii) Hallar los números de almacenes que tienen todos los artículos que incluyen el material con código 123.
viii) Hallar los nombres de los proveedores de todos los materiales que componen el artículo A303.
ix) Hallar los proveedores (código) que proveen al menos un material para todo artículo.
x) Hallar los códigos y descripciones de los artículos que tienen al menos un material que no provee ningún proveedor.
xi) Hallar para cada almacén el precio mínimo, máximo y promedio de los artículos que tiene.
xii) Listar para cada almacén el stock valorizado (código y descripción de cada artículo, cantidad, precio unitario, valor total del artículo).
xiii) Listar el stock valorizado agregado (independiente del almacén) para todos los artículos cuya existencia supera 100 unidades.
xiv) Hallar los artículos cuyo precio es superior a $50 y que están compuestos por más de tres materiales.
xv) Listar los materiales que componen los artículos cuyo precio es superior al precio promedio de los artículos del almacén nro. 2.

E2 Dada la siguiente base de datos

Proveedor (nro_p, nom_p, categoría, ciud_p)
Artículo (nro_a, descripción, ciud_a, precio)

Cliente (nro_c, nom_c, ciud_c)

Pedido (ped_nro, nro_p, nro_a, nro_c, cantidad, precio_total)

Responder a las siguientes consultas en SQL:

i. Hallar el código (nro_p) de los proveedores que proveen el artículo a146.
ii. Hallar los clientes(nom_p) que solicitan artículos provistos por p015.
iii. Hallar los clientes que solicitan algún ítem provisto por proveedores con categoría mayor que 4.
iv. Hallar los ítem pedidos por clientes de Rosario.
v. Hallar los pedidos en los que un cliente de Rosario solicita artículos producidos en la ciudad de Mendoza.
vi. Hallar los pedidos en los que el cliente c23 solicita artículos no solicitados por el cliente c30.
vii. Hallar los pares de ciudades en la forma (ciudad1, ciudad2), tales que un proveedor en ciudad1 provea artículos pedidos por clientes de ciudad2.
viii. Hallar los nombres de los proveedores cuya categoría sea mayor que la de todos los proveedores que proveen el artículo "cuaderno".
ix. Hallar los proveedores que proveen el artículo más caro comprado alguna vez por un cliente de la ciudad de Mendoza.
x. Hallar los clientes que han pedido dos o más artículos distintos.
xi. Hallar los proveedores que no tienen ningún pedido en los que el cliente es de la ciudad de Mendoza y el artículo es producido en San Juan.
xii. Hallar los clientes que compran artículos cuyo precio promedio es superior a $100.
xiii. Hallar los proveedores que venden todos los artículos cuyo precio es superior al precio promedio de los artículos que se producen en La Plata.
xiv. Hallar la cantidad de artículos diferentes provistos por cada proveedor que provee a todos los clientes de Junín.
xv. Hallar los proveedores que han provisto más de 1000 unidades entre los artículos A001 y A100.

E3. Sean los siguientes esquemas de relación:

Vuelo (Nro_vuelo, Desde, Hasta)

Avión_utilizado (nro_vuelo, tipo_avión, nro_avión)

Info_pasajeros (nro_vuelo, documento_identidad, nombre, origen, destino, fecha)

Los vuelos no pueden tener más de dos escalas y no hay cambio de tipo de avión para un mismo vuelo.

Realizar las siguientes consultas en SQL:

i. Hallar los números de vuelo desde el origen A hasta el destino F.
ii. Hallar los tipos de avión que no son utilizados en ningún vuelo que pase por B.
iii. Hallar los pasajeros y números de vuelo para aquellos pasajeros que viajan desde A a D pasando por B.
iv. Hallar los tipos de avión que pasan por C.
v. Hallar la cantidad de aviones que figuran en la bd.
vi. Hallar, para cada avión, la cantidad de vuelos distintos en que se encuentra registrado.
vii. Listar los distintos tipo y nro de avión que tienen a H como destino.
viii. Hallar los pasajeros que han volado en todos los aviones de la compañía.
ix. Hallar los pasajeros que han volado más frecuentemente en el último año.
x. Hallar los pasajeros que han volado la mayor cantidad de veces posible en un B-777.
xi. Hallar los pasajeros que han volado a Toronto en fechas más próximas posible.
xii. Hallar los pasajeros que han volado todos los tipos de avión en los últimos 2 años.
xiii. Hallar los aviones que han transportado más veces al pasajero más antiguo.
xiv. Hallar los pasajeros que han realizado una cantidad de vuelos den-

tro del 10% en más o en menos del promedio de vuelos de todos los pasajeros de la compañía.

xv. Listar la cantidad promedio de pasajeros transportados por los aviones de la compañía, por tipo de avión.

E4. Sea el siguiente esquema de base de datos relacional:

Empleado (Nro_Empleado, Nombre, Domicilio, Localidad, Antigüedad)
Función (Nro_Función, Descripción)
Planta (Nro_Planta, Descripción, Localidad)
Tarea (Nro_Empleado, Nro_Función) /* funciones que cada empleado está en condiciones de realizar */
Organización(Nro_planta, Nro_Función) /* funciones previstas como necesarias para el funcionamiento de la planta */
Trabaja (Nro_Empleado, Nro_Planta, Fecha_Desde, Fecha_Hasta)
Distancia (Localidad, Localidad, Distancia) /* todas las combinaciones posibles entre localidades distintas */

Los empleados que trabajan actualmente en cada planta tienen null en Fecha_Hasta.

Realizar las siguientes consultas en SQL:

i. Hallar los nombres de los empleados que no trabajan en Rosario.
ii. Hallar las funciones que sean desempeñadas por un empleado de Capital Federal y que no sean requeridas en una planta de La Plata.
iii. Hallar las localidades donde residan o estén en condiciones de cumplir funciones empleados con más de 20 años de antigüedad.
iv. Hallar los empleados que estén en condiciones de cumplir la misma función en todas las plantas.
v. Hallar las plantas que contemplan en su organización funciones que cualquier empleado con hasta 10 años de antigüedad esté en condiciones de realizar.

vi. Listar la cantidad total de empleados, el promedio de empleados por planta y, para cada planta en particular, la cantidad de empleados que trabajan en ella.
vii. Hallar los nros. de empleados que trabajan a más de 5 km de donde residen.
viii. Hallar los empleados que viven a una distancia mayor de su lugar de trabajo que la distancia promedio de todos los empleados.
ix. Hallar las plantas donde trabajan empleados con más de 2 años de antigüedad en esa planta.
x. Hallar los empleados que no hayan cambiado de planta.
xi. Listar los empleados por planta que tienen una antigüedad superior a la antigüedad promedio de los empleados actuales de cada planta.

E5. Dados los esquemas de relación del E2, dar una instancia para cada uno de ellos y responder a las siguientes consultas, mostrando la tabla resultado y utilizando el operador que en cada caso se indique:

i. Listar los códigos y nombres de todos los clientes, junto con los artículos y cantidad pedidos. Utilizar *left outer join*.
ii. Listar código y nombre de todos los proveedores junto con los códigos de los clientes a quienes les vendieron. Utilizar *right outer join*.
iii. Listar los nombres de todos los clientes, los artículos y cantidad pedidos pero de manera que aparezcan todos los artículos, se hayan vendido o no. Usar *full outer join*.
iv. Listar los nombres de las ciudades en las que se encuentre al menos un cliente o un proveedor (cada ciudad debe aparecer una sola vez en el resultado). Usar *Union*.
v. Listar los nombres de todos los proveedores primero, y de todos los clientes a continuación, teniendo en cuenta que algunos proveedores son también clientes y se desea que aparezcan todos. Usar *Union All*.

E6. Sean los esquemas de relación del E2

i. Escribir una consulta SQL para contar los proveedores que residen en ciudades distintas de La Plata y Bahía Blanca. Asumir que algunos proveedores tienen un valor nulo en ciudad.

ii. Supongamos se agrega una columna cód_postal a la tabla Proveedor. Los proveedores son 150. Inicialmente, el sgbd completa la columna con nulos. Dada la consulta

Select count (cod_postal) from Proveedores

a) ¿Cuál es el resultado de la consulta?

b) Si se actualizan 15 proveedores con sus códigos postales ¿cuál será ahora el resultado?

c) Complete la consulta para permitir obtener la cantidad total de proveedores.

iii. ¿Cuál es el resultado de la siguiente consulta

```
Select nom_p  From Proveedor p  Where not exists (
      Select *  from Proveedor q where  ciu_p = 'Mendoza' and
                        q.categoria = p.categoria);
```

si existe un único proveedor en Mendoza y su categoría es null?

E7. Sea el siguiente esquema de base de datos relacional:

Empleado (Nro_Empleado, Nombre, Localidad, Antigüedad, Sueldo)

Función (Nro_Función, Descripción)

Planta (Nro_Planta, Descripción, Localidad, Gastos)

Tarea (Nro_Empleado, Nro_Función) /* funciones que cada empleado está en condiciones de realizar */

Organización(Nro_planta, Nro_Función) /* funciones previstas como necesarias para el funcionamiento de la planta */

Trabaja (Nro_Empleado, Nro_Planta, Fecha_Desde, Fecha_Hasta)

Distancia (Localidad1, Localidad2, Distancia) /* todas las combinaciones posibles entre localidades distintas */

Los empleados que trabajan actualmente en cada planta tienen null en Fecha_Hasta.

Utilizar SQL para crear las tablas correspondientes a las relaciones anteriores, con las restricciones de integridad que en cada caso se soliciten. Se deberá dar nombres a todas las restricciones que se definan.

i. Crear todas las tablas, especificando en cada caso las claves primarias y foráneas.

ii. En la tabla Empleado asegurarse que: a) el sueldo esté en el rango 500-5500, b) la antigüedad esté en el rango 0 – 47 (usar un dominio definido en ese rango), y c) nombre y localidad sean no nulos.

iii. En la tabla Planta asegurarse que localidad sea no nulo.

iv. Para la tabla tarea especificar que si el empleado es dado de baja en la tabla Empleado, luego todos sus registros en Tarea son eliminados. Si el empleado cambia de número (Nro_Empleado), también todos sus registros en Tarea serán actualizados.

v. Para la tabla Organización especificar que si tanto una planta como una función son dados de baja, sus correspondientes tuplas serán eliminadas.

vi. Para la tabla Trabaja incluir restricciones que aseguren que:

a) si el empleado o la planta son dados de baja, también lo serán sus tuplas en Trabaja;

b) Fecha_Hasta es nulo o Fecha_Hasta > Fecha_Desde.

vii. Para la tabla Distancia:

a) Localidad1 y Localidad2 deben ser localidades que existan en la tabla Empleados o en la tabla Planta.

b) La distancia debe ser mayor que 0 menor que 700.

c) Los pares de ciudades, independientemente del orden, no deben repetirse ((B. Blanca, La Plata) es igual a (La Plata, B. Blanca)).

viii. El total de sueldos de los empleados más el total de gastos de las plantas no debe superar $ 10.000.000.

E8. Dados el siguiente esquema de relación

BOM (Componente, Subcomponente, Cantidad) /* que subcomponente integra un componente y en qué cantidad */

Los componentes de mayor nivel son los productos (terminados) y cada producto puede tener, a lo sumo, 4 niveles de subcomponentes.

Se debe elaborar un programa, en algún lenguaje de programación tal co-

mo "C", a su elección, que utilice SQL Embebido para responder a las siguientes consultas:

i. Listar los productos terminados
ii. Listar para cada producto los subcomponentes y cantidad insumida de cada uno.
iii. Listar, a elección del usuario, la descomposición de un producto, indicando subcomponentes y cantidades.
iv. Hallar el producto que mayor cantidad de subcomponentes tenga, entre todos los niveles
v. Listar todos los productos en que se halla incluido un subcomponente dado, a elección del usuario
vi. Dado un pedido de un cliente, en el que se indican productos y cantidades, determinar los subcomponentes necesarios y la cantidad requerida de cada uno para su elaboración

NOTAS BIBLIOGRÁFICAS

Los primeros artículos referidos a SQL son los publicados en relación con el desarrollo del prototipo System R de IBM: D. D. Chamberlin y otros [53] y M. M. Astraham y D. D. Chamberlin [10].

Existen numerosos textos sobre SQL en particular, de los cuales mencionaremos el de J. Melton y A. Simon [156] ya que realiza un extenso tratamiento de SQL-92. Entre los textos sobre bases de datos en general, se distinguen por su presentación detallada de SQL R. Ramakrishnan y J. Gehrke [174], P. O'Neil [164] y C. J. Date [71]. J. D. Ullman y J. Widom [203] utilizan SQL-92 e introducen características de SQL3 (ahora SQL:1999), el inminente futuro estándar.

SQL-92, según el documento ANSI X3.135-1992, es el estándar actual adoptado por ANSI (American National Standards Institute) y por ISO (International Organization for Standarization). Para una descripción de SQL:1999 referirse al Apéndice.

Sobre seguridad e integridad ver E. B. Fernández, R. C. Summers y C. Wood [90] y el más reciente S. Castano y otros [51].

CAPÍTULO 5

TEORÍA DEL DISEÑO RELACIONAL

Hasta el momento, hemos supuesto que la estructura de una base de datos relacional está dada de antemano. En este capítulo discutiremos cómo un diseñador puede generar esa estructura de modo de evitar ciertos problemas lógicos tales como redundancia o pérdida de información. Una de las herramientas principales para facilitar esta tarea son restricciones sobre los posibles valores que la información en la base de datos puede tomar. Por ejemplo, el concepto de clave es un caso especial de tales restricciones: no existen dos alumnos con el mismo *número*, o dos libros con el mismo autor y el mismo título. En este capítulo estudiaremos en particular las restricciones llamadas *dependencias funcionales* que generalizan el concepto de clave, como así también otros tipos de dependencias, y su uso en la obtención de *formas normales*.

5.1. ESQUEMAS INDESEABLES

Supongamos que nuestro esquema de base de datos consiste de tres esquemas de relación:

Alumno (Anúmero, Anombre),
Examen (Anúmero, Materia, Efecha, Nota),
Inscrip (Anúmero, Materia, Ifecha).

Una tupla $<n, m, f, x>$ en *Examen* indica que el alumno número n rindió examen en la materia m en la fecha f y obtuvo la nota x. Una tupla $<n, m, f>$

en la relación *Inscrip* indica que el alumno número *n* se inscribió en la materia *m* en la fecha *f*. ¿Cómo puede un diseñador llegar a generar este esquema? Veamos algunas alternativas y sus posibles desventajas.

Redundancia

Dado un conjunto de datos que se desea representar, tales como nombres y números de alumnos, resultados de exámenes, inscripciones, etc., sería lo más simple tomar todos los atributos y formar un solo esquema de relación:

Univ (Anúmero, Anombre, Materia, Ifecha, Efecha, Nota)

Sea *(Anúmero, Materia, Efecha)* la clave primaria de este esquema.

Supongamos que la alumna Isabel Carrizo, número 10203229, ha cursado 25 materias y rendido examen en todas ellas, algunas más de una vez. Cada examen rendido requiere una tupla en *Univ*, y cada una de estas tuplas repite innecesariamente el nombre "Isabel Carrizo", cuando bastaría con el número. Si Isabel se inscribió una sola vez en Álgebra II, en 1994, pero la rindió 18 veces, el dato de que se inscribió en 1994 se repite innecesariamente en 18 tuplas. Por un lado, esto implica espacio desperdiciado en el almacenamiento de la relación. Por otro lado, la redundancia dificulta la actualización. Por ejemplo, si Isabel se casa y quiere que todos sus datos sean modificados a su nombre de casada, tenemos que modificar una tupla por cada examen que ella haya rendido. La esencia del problema es que estamos juntando en una misma relación hechos que son independientes entre sí: el nombre de la alumna número 10203229 es independiente de los exámenes rendidos, y la fecha de examen es independiente de la fecha de inscripción.

Anomalías de Actualización

Volvamos a considerar el esquema *Univ* de la sección anterior, y supongamos que queremos agregar el dato de que en la materia Introducción al Crochet se tomó examen el día 18 de junio de 1998, pero ningún alumno se presentó. No podemos representar esta información mediante una tupla en

la relación *Univ*, ya que ésta requiere valores de nombre y número de alumno, etc. Una posibilidad sería usar algún tipo de *valores nulos* en los atributos no usados, o sea:

<_ _ "Introducción al Crochet", _18/06/98, _>

Pero esto requiere un valor nulo para parte de la clave primaria *(Anúmero, Materia, Efecha)*, lo cual no es aceptable. Hay otros problemas conceptuales asociados con el uso de valores nulos que hacen que se trate generalmente de minimizar su uso. Este tipo de problemas en la actualización se llaman "anomalías de actualización".

Pérdida de Información

Las dificultades notadas arriba sugieren que el problema surge de agrupar en un mismo esquema de relación datos independientes. Por lo tanto, es razonable pensar en *descomponer* un esquema de relación cuando esto ocurre. Pero no cualquier descomposición es aceptable.

Supongamos, por ejemplo, que decidimos descomponer la relación *Univ* en varias relaciones "más pequeñas":

Alumno (Anúmero, Anombre)
Resultado (Anúmero, Materia, Nota)
Examen (Materia, Efecha)
Inscrip (Anúmero, Materia, Ifecha)

Cada una de las relaciones correspondientes se obtienen mediante una proyección de la relación *Univ*:

$$Alumno = \pi_{Anúmero,Anombre}(Univ)$$
$$Resultado = \pi_{Anúmero,Materia,Nota}(Univ)$$
$$Examen = \pi_{Materia,Efecha}(Univ)$$
$$Inscrip = \pi_{Anúmero,Materia,Ifecha}(Univ)$$

Consideremos la consulta "¿en qué fechas dio exámenes el alumno 6017512?" Como los atributos *Anúmero* y *Efecha* no aparecen juntos en nin-

gún esquema de relación, vamos a tener que realizar una operación de junta; parece razonable probar

Resultado |×| *Examen*

y seleccionar del resultado las tuplas con *Anúmero = 6017512*. Sin embargo, es claro que esta expresión no es correcta. Si el alumno 6017512 rindió Topología Cuántica en diciembre de 1996, pero la relación *Examen* indica otras 100 fechas distintas en que se tomó examen de Topología Cuántica, el resultado de la junta va a asociar ese alumno con esas 100 fechas. Lo mismo ocurrirá con todas las materias que haya rendido el alumno. Paradójicamente, a pesar de que la respuesta tiene muchas más tuplas que la respuesta correcta, hemos *perdido* información: el dato específico de que un alumno rindió un examen en una cierta fecha se ha perdido al hacer esta descomposición. Esto es un ejemplo de lo que se llama *descomposición con pérdida*.

Examinando los dos esquemas *Resultado* y *Examen* observamos que el problema es que la conexión entre materias y fechas almacenada en *Examen* no es independiente de la conexión entre alumnos y materias almacenada en *Resultado*, pero la operación de junta se comporta como si lo fueran. No es que la dificultad provenga del uso del operador de junta: se puede demostrar en general que ningún otro operador relacional puede reconstruir la información que se ha perdido en esta descomposición.

Comparemos esta descomposición a pérdida con la que introdujimos al principio del capítulo:

Alumno (Anúmero,Anombre)
Examen (Anúmero, Materia, Efecha, Nota)
Inscrip (Anúmero, Materia)

Supongamos que queremos saber ahora en qué fechas dio examen Isabel Carrizo. Nuevamente necesitamos una junta; en este caso es natural usar

Alumno |×| *Examen*

y seleccionar del resultado aquellas tuplas con *Anombre = Isabel Carrizo*. En este caso la junta no genera ninguna tupla espúrea. ¿Cuál es la diferencia con el caso anterior? Es que los datos almacenados en el esquema *Alumno* son independientes de los almacenados en el esquema *Examen*, ya que el nombre de un alumno con un cierto número no tiene nada que ver con los exámenes rendidos por ese alumno. Para llegar a esta conclusión hemos tenido que usar conocimiento derivado de nuestra familiaridad con el mundo exterior que esta base de datos representa. El objetivo de la teoría de diseño relacional es formalizar este tipo de conocimiento y descubrir métodos formales para analizar esquemas de base de datos usando estos conocimientos. En nuestra definición del modelo relacional, ya hemos presentado un concepto que es un primer paso en esta dirección: el concepto de clave. Para ver cómo el análisis de claves puede ser útil en determinar si una descomposición produce o no pérdida de información, consideremos la intersección de los dos esquemas de relación, *Alumno* ∩ *Examen* = *Anúmero*. Observamos que *Anúmero* es una clave de *Alumno*, ya que para cada número de alumno existe un único nombre. Esto implica que la conexión entre número de alumno y nombre es independiente de cualquier otro dato: de lo contrario tendría que ser posible que el mismo número apareciera con más de un nombre. Por lo tanto, podemos concluir que la junta de *Alumno* y *Examen* no produce pérdida de información.

Con esta motivación, en la sección siguiente pasamos al estudio de las *dependencias funcionales*, una generalización de los conceptos de clave y superclave.

5.2. DEPENDENCIAS FUNCIONALES

Una *dependencia funcional* es una restricción sobre el conjunto de tuplas que pueden aparecer en una relación. Sea R un esquema de relación, y X, Y dos subconjuntos de R. La dependencia funcional *(df)*

$$X \rightarrow Y$$

(leída "X determina Y") se verifica en R si en ninguna instancia aceptable de R pueden existir dos tuplas t_1, t_2 tales que

$$t_1[X] = t_2[X]$$

y

$$t_1[Y] \neq t_2[Y]$$

O sea, dos tuplas que tienen los mismos valores en las columnas X tienen necesariamente que tener los mismos valores en las columnas Y. Notemos que decir que X es una superclave de R es equivalente a decir que $X \rightarrow R$.

No toda dependencia funcional tiene una superclave a la izquierda de la flecha: por ejemplo, en el esquema

Univ (Anúmero, Anombre, Materia, Ifecha, Efecha, Nota),

aunque *Anúmero* no es una superclave, se cumple la df

Anúmero → Anombre

Si suponemos que un alumno se inscribe en una materia una sola vez, también tenemos la df

(Anúmero,Materia) → Ifecha

que es usual escribir

Anúmero, Materia → Ifecha

Una aplicación del concepto de dependencia funcional es la formalización de la noción de clave.

Dado un esquema $R(A_1, \ldots, A_n)$ y un conjunto de dfs F asociado, se dice que $X \subseteq R$ es una *clave* para el esquema R si se cumplen las dos condiciones siguientes:

1. F implica $X \rightarrow A_1 \ldots A_n$ o abreviadamente $X \rightarrow R$.
2. No existe ningún $Z \subseteq X$ tal que F implica $Z \rightarrow R$ (condición de minimalidad).

Por otra parte Y es una superclave para R si Y es un superconjunto de alguna clave o, equivalentemente, cumple la condición 1.

Implicación de Dependencias Funcionales

Dado un conjunto de dfs F que debe cumplirse en una relación, se sigue que esta relación debe necesariamente cumplir otras dfs que pueden no estar en F. Por ejemplo, supongamos que F consiste de las dos dfs

Anúmero → Anombre

y

Anúmero Materia → Ifecha

la papa!

O sea, dos tuplas con el mismo número de alumno deben tener el mismo nombre, y dos tuplas con el mismo número de alumno y la misma materia deben tener la misma fecha de inscripción. Esto implica obviamente que dos tuplas con el mismo número de alumno y la misma materia deben tener el mismo nombre y la misma fecha de inscripción, o sea:

Anúmero Materia → Anombre Ifecha

En general, diremos que un conjunto de dfs F *implica* una df d si toda relación que satisface las dfs de F debe necesariamente satisfacer d. Para poder analizar un esquema de bases de datos usando las dependencias funcionales, es importante poder determinar algorítmicamente si F implica d, dados cualquier conjunto F y df d. El primer paso para obtener tal algoritmo fue dado por W.W. Armstrong en 1974. Armstrong propuso tres reglas de inferencia, o sea, reglas que permiten inferir dfs a partir de un conjunto dado en forma puramente formal. Estas reglas se llaman *axiomas de Armstrong* y son:

1. *Reflexividad*: Si $Y \subseteq X$, inferir $X \rightarrow Y$.
2. *Aumento*: Para cualquier conjunto W, de $X \rightarrow Y$ inferir $XW \rightarrow YW$.
3. *Transitividad*: De $X \rightarrow Y$ e $Y \rightarrow Z$, inferir $X \rightarrow Z$.

EJEMPLO 5.1: consideremos el esquema $R(A,B,C,D,E)$ y sea F el conjunto de dfs

$$AB \rightarrow C$$
$$C \rightarrow D$$
$$BD \rightarrow E$$

La siguiente es una derivación de la df $AB \rightarrow E$ usando los axiomas:

1	$AB \rightarrow C$	(dada)
2	$C \rightarrow D$	(dada)
3	$AB \rightarrow D$	(transitividad de 1,2)
4	$AB \rightarrow BD$	(aumento de 3)
5	$BD \rightarrow E$	(dada)
6	$AB \rightarrow E$	(transitividad de 4,5)

Dado un conjunto de dfs F, sea la *clausura* de F, denominada F^+, el conjunto de todas las dfs que pueden inferirse de F usando los axiomas de Armstrong. Simbólicamente

$$F^+ = \{X \rightarrow Y \mid F \models X \rightarrow Y\}$$

Se puede demostrar que los axiomas son *válidos y completos*. Que son *válidos* significa que toda df en F^+ es realmente implicada por F; que son *completos* significa que toda df implicada por F pertenece a F^+. O sea, la validez de los axiomas garantiza que no nos harán llegar a conclusiones falsas, y la completitud afirma que nos permitirán obtener todas las conclusiones correctas.

Una df $X \rightarrow Y$ se denomina *trivial* si dado cualquier R con $X, Y \subseteq R$, la dependencia se satisface en cualquier instancia r de R. No es difícil ver que las dfs triviales son exactamente aquellas en que $Y \subseteq X$ (¿por qué?).

Con el fin de facilitar el proceso de derivación de dfs es posible definir un conjunto de reglas de inferencia adicionales a los Axiomas de Armstrong. Estas reglas se enuncian a continuación y pueden ser probadas fácilmente utilizando los axiomas.

Reglas de Inferencia Adicionales

RI1 Regla de Unión: de $X \rightarrow Y$ y $X \rightarrow Z$, inferir $X \rightarrow YZ$.

RI2 Regla de Pseudotransitividad: para cualquier W, de $X \rightarrow Y$ e $YW \rightarrow Z$ inferir $XW \rightarrow Z$.

RI3 Regla de Descomposición: de $X \rightarrow YZ$ inferir $X \rightarrow Y$ y $X \rightarrow Z$.

A modo de ejemplo complementario probaremos la regla de unión:

1	$X \rightarrow Y$	(dada)
2	$X \rightarrow Z$	(dada)
3	$X \rightarrow XY$	(aumento de 1 con X) [Obsérvese $XX = X \cup X = X$]
4	$XY \rightarrow YZ$	(aumento de 2 con Y)
5	$X \rightarrow YZ$	(transitividad de 3,4)

Clausura de un Conjunto de Atributos

Dado un esquema de relación R, un conjunto de dfs F y un conjunto de atributos $X \subseteq R$, se define la *clausura de X* con respecto a F, indicada por X_F^+, como el conjunto de todos los atributos $A \in R$ tal que $X \rightarrow A$ puede ser derivado desde F utilizando los axiomas (y las reglas adicionales) de todas las formas posibles. Simbólicamente:

$$X_F^+ = \{A \in R \mid F \models X \rightarrow A\}$$

Este concepto es de gran importancia en la resolución del problema de la implicación de dependencias funcionales y resulta ampliamente usado en los algoritmos relacionados con la normalización de esquemas relacionales. A continuación presentaremos un algoritmo para el cálculo de XF^+ (en adelante mientras no surja ambigüedad omitiremos el subíndice F).

Algoritmo 1: Clausura de X con respecto a F: X^+

Entrada: un conjunto de dfs F y un conjunto de atributos X

Salida: X^+, la clausura de X con respecto a F

```
1   Cl := X;
2   Cl' := ∅;
3   while Cl <> Cl' do begin
4       Cl' := Cl;
5       for each df V → W ∈ F do
6           if V ⊆ Cl then begin
7               Cl := Cl ∪ W;
8           end (if)
9   end (while)
10  X+ := Cl;
```

El Algoritmo 1 computa X^+ en Cl partiendo de X, ya que siempre $X \subseteq X^+$ por el Axioma de Reflexividad. Luego hace una o más pasadas sobre las dependencias de F y para cada una de ella verifica si el lado izquierdo (li) está incluido en Cl. Si es así, agrega el lado derecho (ld) a Cl. El algoritmo corre en $O(n^2)$, donde n depende de la cantidad de df's de F y la cantidad de atributos. Sin embargo existe un algoritmo debido a Beeri y Bernstein que corre en tiempo lineal.

Veamos ahora un ejemplo de aplicación del Algoritmo 1:

EJEMPLO 5.2: sean R y F como en el ejemplo anterior. Deseamos computar $(AB)^+$. Recordemos que $F = \{AB \rightarrow C, C \rightarrow D, BD \rightarrow E\}$.

1. $Cl = AB$
2. $Cl' = \varnothing$
3. $Cl' = AB$
4. $Cl = AB \cup C$, al considerar $AB \rightarrow C$ y $AB \subseteq Cl$
5. $Cl = ABC \cup D$, al considerar $C \rightarrow D$ y $C \subseteq Cl$
6. $Cl = ABCD \cup E$, al considerar $BD \rightarrow E$ y $BD \subseteq Cl$
7. $Cl' = Cl$

Como no se produce ningún cambio en Cl, el algoritmo sale del while y termina con

8. $X^+ = ABCDE$

El siguiente es un importante resultado relacionado con las aplicaciones de X^+:

es implicado

(1) *Sea F un conjunto de df's y $X \rightarrow Y$ una df, luego $F \models X \rightarrow Y$ si y sólo si $Y \subseteq X^+$.*

Por lo tanto, si nos dan una df $X \rightarrow Y$ y deseamos saber si es implicada por F, o equivalentemente si $X \rightarrow Y \in F^+$, sólo necesitamos calcular X^+ y verificar si $Y \subseteq X^+$. Otra aplicación de este resultado, relacionada con la determinación de las claves de un esquema, es la siguiente: supongamos que tenemos la sospecha (o simplemente deseamos verificar) que un determinado conjunto de atributos $X \subseteq R$ es una clave para R. Luego computamos X^+ y comprobamos si $X^+ = R$. Si es así, vemos que se satisface el lado derecho de la doble implicación de (1) donde R juega el papel de Y, entonces también se cumple que $F \models X \rightarrow R$, por lo que X resulta una superclave.

Clave

En el ejemplo anterior $X = AB$ es una superclave para $R(A, B, C, D, E)$. Si

quisiéramos saber si X es una clave, o sea cumple también la condición de minimalidad, deberíamos verificar si $A^+ = R$ o $B^+ = R$. En general, deberían computarse las clausuras de los subconjuntos de X buscando los $Z \subseteq X$ más pequeños que cumplen $Z^+ = R$. En nuestro caso, AB es también clave.

$A^+ = A$, $B^+ = B \Rightarrow AB$ es determinante

Equivalencia de Conjuntos de Dependencias Funcionales

En el mundo real, el conjunto de df's F es una abstracción de las reglas que rigen las relaciones entre los componentes de la organización para la cual se diseña, construye y opera una base de datos. Luego F representa un conjunto de restricciones que, de algún modo, dice cuáles pueden ser los datos corrientes de la bd. Podríamos pensar entonces que ese conjunto de restricciones puede estar representado por diversos conjuntos de df's, por ejemplo F y G. El siguiente interrogante, lógicamente, sería saber si F y G son equivalentes y qué significa exactamente que lo sean. Para ello definimos:

Dados un esquema R, F y G conjuntos de df's, F cubre a G si $F \models G$. Si además G cubre a F, luego F y G son equivalentes y escribimos $F \equiv G$.

Otra forma de expresar lo anterior es: *F y G son equivalentes si y sólo si $F^+ = G^+$.*

Para probar que dos conjuntos F y G son equivalentes, sin necesidad de calcular F^+ ni G^+, se debería probar que toda dependencia $X \rightarrow Y \in F$ es implicada por G y, recíprocamente, toda dependencia de G es implicada por F. Esto puede hacerse fácilmente utilizando el Algoritmo 1.

Los conjuntos equivalentes de df's expresan exactamente lo mismo, pero pueden diferir en ciertas características como el tamaño. En general, buscaremos conjuntos de df's equivalentes pero que faciliten el trabajo de los algoritmos de normalización.

Cubrimiento Minimal para un Conjunto de df's

Dado un conjunto de df's F, deseamos hallar un conjunto F_M equivalente a F, pero con la condición de que sea *minimal*, en el sentido de que no haya atributos redundantes en las dependencias ni tampoco dependencias redundantes. Decimos que F_M es un *Cubrimiento Minimal* para F si $F_M \equiv F$ y cumple las siguientes tres condiciones:

esto decido

i) Todo ld de una df de F_M tiene un único atributo.
ii) Todo li de una df en F_M es reducido, en el sentido de no contener atributos redundantes. Lo anterior se puede expresar como:

$$\forall X \to A \in F_M \neg\exists Z \subset X \text{ tal que } (F_M - \{X \to A\} \cup \{Z \to A\}) \equiv F_M$$

iii) F_M no contiene df redundantes, es decir, $\neg\exists X \to A$ tal que

$$F_M - \{X \to A\} \equiv F_M$$

El siguiente algoritmo permite hallar un cubrimiento minimal para un conjunto de df's F dado.

Algoritmo 2: hallar un cubrimiento minimal F_M para F.
Entrada: conjunto F de dependencias funcionales.
Salida: conjunto F_M.

Procedimiento:
Se realiza en tres pasos, donde cada uno se ejecuta una única vez, en forma completa.

Paso 0: $F_{M0} := F$;

Paso 1 – Descomposición: reemplazar cada df de la forma

$$X \to A_1, \ldots, A_n \in F_{M0}$$

por las dependencias $X \to A_1, \ldots, X \to A_n$.
El resultado de este paso lo denominamos F_{M1} el que, en virtud de la regla de descomposición, resulta

$$F_{M1} \equiv F_{M0}$$

Paso 2 – Eliminación de atributos redundantes:

```
i      F_M2 := F_M1;
ii     for each dependencia X → A ∈ F_M1 do begin
iii        Z := X;
iv         for each atributo B ∈ X do
v            if A ∈ (Z − B)+_FM1 and A ∈ (Z − B)+_FM2−{Z→A}∪{Z−B→A}
             then
vi             Z := Z − B;
vii        reemplazar X → A por Z → A;
viii   end.
```

evito el clausurar en cuanto

es la vez quitando atributos

y: ABC
AB su clausura
A

Es claro que estos cambios preservan la equivalencia, por lo que al finalizar el Paso 2 resulta $F_{M2} \equiv F_{M1}$.

Paso 3 – Eliminación de dependencias redundantes: $\{\cancel{B \to C}, \cancel{B \to D}, B \to A, C \to D, D \to A, D \to E\}$

```
i      F_M := FM_2;
ii     for each dependencia X → A ∈ F_M do begin
iii        G := F_M − {X → A};
iv         if A ∈ X+_G then
v              F_M := G;
vi     end.
```

$B \to C$
¿ $C \in B^+_G$?
$B^+_G = BDAE$ no

$B \to D$
¿ $D \in B^+_G$
$B^+_G = EADB$
$D \in B^+_G$
$\Rightarrow$ Es ~~sin~~ $F_M - \{B$

Los cambios realizados en este paso también preservan la equivalencia por lo que $F_M \equiv F_{M2}$ y, en virtud de lo expresado al final de cada paso resulta $F_M \equiv F$.
Veamos ahora un ejemplo para clarificar estos conceptos:

EJEMPLO 5.3: Sea $F = \{AB \to CD, B \to A, C \to D, D \to A, BD \to E\}$

Paso 1: aplicamos descomposición a $AB \to CD$, por lo que reemplazamos esa dependencia por $AB \to C$ y $AB \to D$ resultando

$$F_{M1} = \{AB \to C, AB \to D, B \to A, C \to D, D \to A, BD \to E\}$$

Paso 2: comenzamos tratando de eliminar atributos redundantes en $AB \to C$. Veamos si A es redundante: se debe cumplir que $C \in B^+_{FM1}$. Aplicando el Algoritmo 1 obtenemos $B^+_{FM1} = ABCDE$. También se debe cumplir que $C \in B^+$ con respecto a $F_{M2} - \{AB \to C\} \cup \{B \to C\}$. Nuevamente $B^+ = ABCDE$, con lo que podemos reemplazar $AB \to C$ por $B \to C$. De la misma forma reemplazamos $AB \to D$ por $B \to D$ y $BD \to E$ por $B \to E$.
De esta manera queda

las dependencias

$$F_{M2} = \{B \to C, B \to D, B \to A, C \to D, D \to A, B \to E\}$$

Paso 3: La primera dependencia que hallamos y que se puede eliminar es $B \to D$, ya que es posible obtenerla por transitividad entre $B \to C$ y $C \to D$. La segunda y última es $B \to A$, la que también puede derivarse por transitividad, primero entre $B \to C$ y $C \to D$, y luego este resultado con $D \to A$. Finalmente nos queda el cubrimiento minimal para F,

$$F_M = \{ B \to C, C \to D, D \to A, B \to E \}$$

Si se implementa el Algoritmo 2 usando el algoritmo de Beeri y Bernstein para el cómputo de la clausura de un conjunto de atributos (recordemos que este algoritmo corre en tiempo lineal), su complejidad computacional resulta de O(n.m), donde n es la cantidad de dfs y m la longitud de los datos de entrada.

5.3. Propiedades Deseables de los Esquemas

Vamos a formalizar la discusión de la sección anterior presentando varias propiedades que queremos imponer a nuestros esquemas de base de datos para evitar las dificultades de redundancia, anomalías de actualización y pérdida de información.

Descomposición sin Pérdida

Sea un esquema de relación $R = \{A_1, \ldots, A_n\}$ con un conjunto de dependencias funcionales F. Una *descomposición* ρ (rho) de R es un conjunto de esquemas de relación

$$\rho = \{R_1, \ldots, R_k\}$$

tales que

$$\cup_{i=1}^{k} R_i = R$$

Decimos que ρ es una descomposición *sin pérdida* si, para cada instancia r de R que satisface las dfs F, se verifica que

$$r = |\times|_{i=1}^{k} \pi_{Ri}(r)$$

Es fácil demostrar que la desigualdad de conjuntos

$$r \subseteq |\times|_{i=1}^{k} \pi_{Ri}(r)$$

se verifica para cualquier r, satisfaga o no las dependencias funcionales. La otra igualdad, en cambio, puede no ser cierta, como lo muestra el ejemplo de *Resultado* |×| *Examen* más arriba. Intuitivamente, esto implica que no podemos reconstruir el valor original de r después de haberla proyectado sobre los R_i's; al intentar efectuar esta reconstrucción mediante una operación de junta, obtendremos tuplas que no pertenecían a la relación original.

Es importante poder determinar, dados R, F, y ρ, si ρ es una descomposición sin pérdida de R. Consideremos el caso especial en que es una descomposición binaria, o sea

$$\rho = \{R_1, R_2\}$$

Para dar una idea del tipo de argumentos usados, vamos a demostrar que si la intersección de los atributos de R_1 y R_2 forma una superclave para uno de los dos esquemas, entonces la descomposición es sin pérdida.

Sea $X = R_1 \cap R_2$, y supongamos que X es una superclave de R_1, o sea, que F implica la df $X \to R_1$. Sea r una instancia de R que satisface F, y sea

$$r' = \pi_{R_1}(r) \,|\times|\, \pi_{R_2}(r)$$

Consideremos una tupla $t \in r'$. Sea $x = t[X]$. Por definición de r', existen tuplas $t_1 \in \pi_{R_1}(r)$ y $t_2 \in \pi_{R_2}(r)$ tales que $t[R_1] = t_1$ y $t[R_2] = t_2$. Notemos que esto implica que $t_1[X]=t_2[X]=x$. Las tuplas t_1 y t_2 provienen de proyectar tuplas de r, o sea, existen dos tuplas $s_1 \in r$ y $s_2 \in r$ tales que $s_1[R_1] = t_1$ y $s_2[R_2] = t_2$. Además, $s_1[X]=s_2[X]=x$. Como r satisface la df $X \to R_1$, se sigue que $s_1[R_1]=s_2[R_1]$, y por las igualdades de más arriba, $t[R_1]=s_2[R_1]$ y $t[R_2]=s_2[R_2]$, o sea, $t = s_2$. Por lo tanto, $t \in r$. Lo que hemos demostrado es que

$$r \supseteq \pi_{R_1}(r) \,|\times|\, \pi_{R_2}(r)$$

Como la otra inclusión es siempre cierta, se sigue la igualdad.

En realidad, aunque no vamos a demostrarlo aquí, también es cierto lo inverso, o sea, si la descomposición es sin pérdida, F debe implicar que $R_1 \cap R_2$ es una superclave para R_1 o para R_2.

¿Qué ocurre cuando la descomposición no es binaria? En este caso no existe una caracterización de las descomposiciones sin pérdida tan simple

como la dada, pero sí existe un algoritmo eficiente para decidir, dados R, F y ρ, si ρ es sin pérdida.

El mencionado algoritmo es una simplificación del Método Chase. Su denominación se aplica al proceso de modificar una relación convenientemente para que satisfaga restricciones dadas por un conjunto de dfs. En particular, el Chase se aplica a una representación tabular de la descomposición conocida en la teoría de bases de datos como tableau.

Dado R un tableau T para una descomposición $\rho = (R_1, \ldots, R_k)$ de R se define de la siguiente forma:

1. T tiene n columnas, una para cada atributo de R.
2. T tiene k filas, una para cada esquema de la descomposición
3. Dadas la fila i y la columna j (esquema R_i y atributo Aj), el contenido del tableau será:

$$a_j \text{ si } A_j \in R_i \text{ o}$$

$$b_{ij} \text{ si } A_j \notin R_i$$

aj: distinguidos

bij = no distinguidos

Los a_j se denominan variables o símbolos distinguidos y son únicos por columna, pudiendo repetirse en diferentes filas. Los b_{ij} se denominan variables o símbolos no distinguidos y son todos diferentes.

El siguiente algoritmo verifica si una descomposición de R es sin pérdida de información.

Algoritmo 3: Prueba si una descomposición es sin pérdida de información
Entrada: Esquema $R(A_1, \ldots, A_m)$, conjunto de dfs F y descomposición $\rho = (R_1, \ldots, R_k)$.
Salida: "Sí" si la descomposición es sin pérdida de información.
"No" en caso contrario.

Procedimiento

1. Construir el tableau T para la descomposición propuesta. Llamémoslo T_0 por ser el tableau inicial.
2. while haya cambios sobre T do begin
/*Considerar a T como una instancia de R. Las variables pueden adoptar valores desde los dominios asociados a las correspondientes columnas */

3. for each df $X \to A \in F$ do
4. for each par de filas t_i, t_h de T do
5. if ($t_i[X] = t_h[X]$) and ($t_i[A] \neq t_h[A]$) then
6. /* aplicamos la regla correspondiente a esa dependencia y modificamos el tableau de la siguiente forma: */
7. i) sin pérdida de generalidad supongamos que $t_i[A]$ contiene a_j (asumimos también que A corresponde a la columna j), luego debemos igualar $t_h[A]$ a a_j.
ii) Supongamos ahora que ambas filas tienen símbolos no distinguidos

$$t_i[A] = b_{ij} \text{ y } t_h[A] = b_{hj}$$

luego igualamos $t_h[A]$ a b_{ij}.

9. end (while)
10. Llamemos T* al tableau final. Si T* contiene al menos una fila con todos sus símbolos a's ($w_d = \langle a_1, a_2, \ldots, a_m \rangle$), es decir, sólo variables distinguidas, entonces emitir "Sí". De lo contrario el resultado a emitir es "No".

Obsérvese que en el paso 5 no hay ninguna restricción para los valores que pueda tomar $t_i[X] = th[X]$: puede ser tanto un conjunto de variables distinguidas como un conjunto de variables no distinguidas o una mezcla de ambas.

EJEMPLO 5.4: Dado un esquema $R = ABCDE$ y un conjunto dfs $F = \{B \to C, C \to D, D \to A, B \to E\}$, deseamos verificar si la descomposición $\rho = (AB, BCD, DE)$ es sin pérdida de información.

Comenzamos construyendo el tableau inicial T_0 para la descomposición

T_0:

	A	B	C	D	E
AB	a_1	a_2	b_{13}	b_{14}	b_{15}
BCD	b_{21}	a_2	a_3	a_4	b_{25}
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

Vemos que T_0 viola la df $B \to C$. Aplicando la transformación correspondiente a esa df obtenemos:

T_1:

	A	B	C	D	E
AB	a_1	a_2	a_3	b_{14}	b_{15}
BCD	b_{21}	a_2	a_3	a_4	b_{25}
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

se reescribió a_3 en lugar de b_{13}

Observamos también que T_1 viola la df $C \to D$. Forzando a T_1 a que satisfaga esa dependencia resulta

T_2:

	A	B	C	D	E
AB	a_1	a_2	a_3	a_4	b_1
BCD	b_{21}	a_2	a_3	a_4	b_{25}
DE	b_{31}	b_{32}	b_{33}	a_4	a_5

se reescribió a_4 en lugar de b_{14}

T2 viola $D \to A$. Al aplicar la transformación correspondiente nos queda

T_3:

	A	B	C	D	E
AB	a_1	a_2	a_3	a_4	b_{15}
BCD	a_1	a_2	a_3	a_4	b_{25}
DE	a_1	b_{32}	b_{33}	a_4	a_5

se reescribió a_1 en lugar de b_{21} y de b_{31}

Aplicamos ahora la transformación correspondiente a $B \to E$, ya que T_3 viola esa dependencia:

T4:

	A	B	C	D	E
AB	a_1	a_2	a_3	a_4	b_{15}
BCD	a_1	a_2	a_3	a_4	b_{15}
DE	a_1	b_{32}	b_{33}	a_4	a_5

se reescribió b_{15} en lugar de b_{25}

Finalmente el tableau T4 no viola ninguna df de F por lo que no se le puede aplicar ninguna nueva transformación. Luego T_4 resulta T^*. Observamos que T^* no contiene la fila w_d de las variables distinguidas. Luego nuestro algoritmo produce el resultado "No", es decir, la descomposición es con pérdida de información.

Veamos ahora un ejemplo positivo:

EJEMPLO 5.5: sean R y F igual que en el ejemplo anterior pero ahora la descomposición es $\rho = (ABC, BE, ACD)$.

T_0:

	A	B	C	D	E
ABC	a_1	a_2	a_3	b_{14}	b_{15}
BE	b_{21}	a_2	b_{23}	b_{24}	a_5
ACD	a_1	b_{32}	a_3	a_4	b_{35}

Con una única pasada de todas las dfs de F, en la misma cantidad de pasos del ejemplo anterior, obtenemos

T^*:

	A	B	C	D	E
ABC	a_1	a_2	a_3	a_4	a_5
BE	a_1	a_2	a_3	a_4	a_5
ACD	a_1	b_{32}	a_3	a_4	b_{35}

Como se ve, T^* contiene la fila w_d por lo que la respuesta del algoritmo ahora es "Sí": la descomposición es *sin pérdida de información*

A continuación analizaremos brevemente la complejidad computacional del Algoritmo 3. Si m es la cantidad de columnas de T, k la cantidad de filas de T y p la longitud de F; luego el tamaño de la entrada resulta $n = O(mk + p)$. Se puede probar que el cómputo de T^* se puede obtener en tiempo $O(n^3)$.

La propiedad de no tener pérdida es una de las metas que fijaremos para nuestros esquemas de base de datos, para evitar los problemas causados por operaciones de junta cuyo resultado es semánticamente incorrecto.

Preservación de Dependencias

Otro de los criterios para evaluar un diseño de un esquema de base de datos se basa en la idea de preservación de las dependencias funcionales. Supongamos que tenemos el esquema relacional

A B C D E

Horario (Anúmero, Materia, Aula, Día, Hora),

con las dependencias funcionales

F^+
Aula Dia Hora Materia

F^+

$$\left.\begin{array}{l} \textit{Aula Día Hora} \rightarrow \textit{Materia} \\ \textit{Anúmero Día Hora} \rightarrow \textit{Aula} \end{array}\right\} DF$$

C DE → B ⇒ CDE → BDE ⇒ ADE → BDE
A DE → C ADE → CDE
A → E

Una tupla <*n*, *m*, *a*, *d*, *h*> en esta relación indica que el alumno número *n* está cursando la materia *m* y que esta materia se dicta los días *d* a la hora *h* en el aula *a*. Las dfs expresan las restricciones de que un alumno no puede estar en dos aulas al mismo tiempo y en un aula no pueden dictarse dos materias al mismo tiempo.

Para evitar la redundancia de repetir el lugar, día y hora de dictado de la materia para cada alumno que la cursa, podríamos usar la siguiente descomposición:

A B

$$\rho \left\{ \begin{array}{l} \textit{Cursa (Anúmero, Materia)} \\ \textit{Sedicta (Materia, Día, Hora, Aula)} \end{array} \right.$$

Supongamos ahora que un alumno se anota en una nueva materia, y queremos verificar que no hay conflictos en su horario, o sea que la df

Anúmero Día Hora → Aula

no se viola. Para hacer esta verificación necesitamos tomar primero la junta de *Curso y Sedicta*. Como la operación de junta es costosa, es deseable poder verificar todas las dfs en forma local, o sea, examinando una sola relación. Esto nos permitirá validar actualizaciones sin necesidad de computar juntas.

Sea R un esquema de relación, F un conjunto de dfs sobre R, y

$$\rho = (R_1, \ldots, R_k)$$

una descomposición de R. Sea F_i el conjunto de todas las dfs en F^+ que mencionan solamente atributos de R_i. Simbólicamente

$$F_i = \{X \rightarrow Y \mid X \rightarrow Y \in F^+ \text{ y } XY \subseteq R_i\},$$

lo que también se suele expresar como

$$F_i = \pi_{R_i}(F)$$

Es claro que, para verificar si la base de datos satisface F_i, necesitamos examinar solamente la instancia de R_i. Si la verificación de cada F_i por separado es suficiente para establecer que F también se verifica, decimos que ρ es una descomposición que *preserva* a F. Más formalmente, ρ preserva a F cuando

$$\boxed{F^+ = (\cup_{i=1}^{k} F_i)^+}$$

→ si se cumple se preserva las dependencias funcionales

Por ejemplo, sea ρ la descomposición *(Cursa, Sedicta)* del ejemplo anterior. F_2 será la clausura de la df

Aula Día Hora → Materia

En cambio, F_1 es la clausura del conjunto vacío de dfs, ya que no hay ninguna df no trivial en el conjunto F^+ que mencione solamente atributos de *Cursa*. Por lo tanto, $(F_1 \cup F_2)^+$ no contiene la df

Aula Día Hora → Materia,

y ρ no preserva a F.

Hemos evitado explicar exactamente cómo se calculan los F_i's a partir de F. Notemos que no basta con seleccionar aquellas dfs en F que mencionan solamente atributos de R_i, sino que es necesario seleccionarlas de F^+, y el cálculo de F^+ es, en general, un proceso de complejidad exponencial. Sin embargo, existe un algoritmo que presentaremos a continuación, que determina en tiempo polinómico si una descomposición preserva un conjunto de dfs.

El algoritmo computa la clausura de un conjunto de atributos con respecto a $\cup_{i=1,\ldots,k} F_i$, sin necesidad de computar F^+. Para esto se define una R_i-operación sobre el conjunto de atributos Z, respecto del conjunto F de dfs, como el reemplazo de Z por $Z \cup ((Z \cap R_i)^+ \cap R_i)$, con la clausura de $Z \cap R_i$ computada con respecto a F. En cada pasada se modifica el valor de Z al aplicar las R_i-operaciones con $i = 1, 2, \ldots, k$. Si al final de una pasada no se produjo ningún cambio sobre Z, el resultado es X^+ con respecto a $\cup_{i=1,\ldots,k} F_i$.

Algoritmo 4: Prueba si una descomposición preserva las dependencias funcionales

Entrada: una descomposición $\rho = (R_1, \ldots, R_k)$ de R y un conjunto de dfs F.
Salida: "Sí" si ρ preserva F
"No" en caso contrario.
Procedimiento:

$Z := X$;
while Z cambie do
for i := 1 to k do
$Z := Z \cup ((Z \cap R_i)^+ \cap R_i)$;

Si Z es el resultado final y, además, $Y \subseteq Z$ luego $X \to Y \in (\cup_{i=1,\ldots,k} F_i)^+$. De manera que si esto se cumple para toda df $X \to Y$ de F el algoritmo emitirá "Sí" (ρ preserva las dependencias) y "No" en el caso contrario.

EJEMPLO 5.6:

sea $R = \{A, B, C, D, E\}$ y
$F = \{AB \to C, B \to D, C \to E, D \to C, E \to B\}$
y $\rho = \{ABC, CD, BDE\}$

$C^+ = \{CEBD\}$

A simple vista se verifica que todas las dfs se preservan, excepto $C \to E$. Por lo tanto sometemos esa df al Algoritmo 4:

$BD^+ = \{BDCE\}$

Comenzamos haciendo $Z = C$.
Como resultado de la ABC-operación obtenemos:

$$C \cup ((C \cap ABC)^+ \cap ABC) = BC$$

$$C \cup (C^+ \cap ABC) = BC$$

Al aplicar ahora la CD-operación resulta:

$$BC \cup ((BC \cap CD)^+ \cap CD) = BCD$$

Finalmente, cuando aplicamos la BDE-operación conseguimos:

$$BCD \cup ((BCD \cap BDE)^+ \cap BDE) = BCDE$$

BD

$\Rightarrow BCDE \cap ABC$
$BCDE \cup BC$

Observamos que si intentáramos una nueva iteración, no conseguiríamos ningún cambio al valor obtenido de $Z = BCDE$. Por lo que sólo nos resta verificar la condición de $Y \subseteq Z$, que evidentemente se cumple, ya que $E \subseteq BCDE$. Luego la descomposición ρ preserva las dfs.

Forma Normal de Boyce-Codd

Hemos mencionado dos de los objetivos de un buen diseño relacional: descomposiciones sin pérdida y preservación de las dependencias funcionales. El tercer objetivo apunta a evitar la redundancia y las anomalías de actualización mediante la definición de una serie de *formas normales*: primera, segunda, y tercera forma normal, forma normal de Boyce-Codd, cuarta forma normal, forma normal de proyección-junta o quinta forma normal. Presentaremos primero la forma normal de Boyce-Codd, la más estricta de las formas normales cuando el único tipo de restricción presente es la dependencia funcional.

Decimos que un esquema de relación R está en formal normal de Boyce-Codd (*FNBC*) si, para toda dependencia funcional $X \to Y$ en F^+, o bien $Y \subseteq X$, o X es una superclave de R. O sea, toda dependencia funcional que no es trivial tiene una superclave como lado izquierdo.

EJEMPLO 5.7:

consideremos el esquema

Examen (Anúmero, Materia, Efecha, Nota).

La única dependencia funcional no trivial es

Anúmero Materia Efecha → Nota.

Como *(Anúmero, Materia, Efecha)* es una superclave de *Examen*, este esquema satisface la FNBC. En cambio, consideremos el esquema

Univ (Anúmero, Anombre, Materia, Ifecha, Efecha, Nota)

Las dependencias funcionales son:

Anúmero → Anombre
Anúmero Materia → Ifecha
Anúmero Materia Efecha → Nota

La única clave es *(Anúmero, Materia, Efecha)*; por lo tanto, toda superclave debe contener estos tres atributos. Las dependencias funcionales *Anúmero → Anombre* y *Anúmero Materia → Ifecha* muestran que este esquema no está en FNBC.

Intuitivamente, la idea de FNBC es la siguiente. Supongamos que $X \rightarrow Y$ es una dependencia funcional no trivial tal que X no es una superclave. Esto significa que el mismo valor de X puede aparecer en múltiples tuplas. Cada vez que se repite un valor de X, debe repetirse también el correspondiente valor de Y para satisfacer $X \rightarrow Y$; de *ahí* la redundancia.

Existen metodologías de diseño que permiten llegar a un esquema en FNBC descomponiendo un esquema universal inicial. Tomemos por ejemplo el esquema

Univ (Anúmero, Anombre, Materia, Ifecha, Efecha, Nota);

Como observamos más arriba, la dependencia

Anúmero → Anombre

muestra que este esquema no está en FNBC. Por lo tanto, lo descomponemos reemplazándolo por

Alumno (Anúmero, Anombre)
Resto (Anúmero, Materia, Ifecha, Efecha, Nota)

En general, la descomposición de R respecto de $X \rightarrow Y$ introduce dos nuevos esquemas $R_1(XY)$, $R_2(X(R-XY))$. Observemos que esta descomposición es sin pérdida dado que X es una superclave de R_1. Ahora aplicamos el mismo proceso recursivamente a R_1 y R_2. En el ejemplo *Alumno* está en FNBC, ya que la única dependencia funcional no trivial en ese esquema es *Anúmero → Anombre*. En cambio, *Resto* tiene la dependencia

Anúmero Materia → Ifecha

y *(Anúmero, Materia)* no es una superclave; por lo tanto *Resto* no está en FNBC. Le aplicamos el proceso de descomposición, obteniendo

Inscrip (Anúmero, Materia, Ifecha)
Examen (Anúmero, Materia, Efecha, Nota)

Éste es el esquema que al principio del capítulo habíamos postulado mágicamente; hemos visto cómo puede obtenerse mecánicamente a partir de un esquema con todos los atributos.

El método que acabamos de describir permite obtener, a partir de un esquema arbitrario, una descomposición sin pérdida en FNBC. Este satisface dos de los objetivos que hemos dado para el diseño de un esquema, pero no el tercero, o sea, la preservación de dependencias. Es más: existen esquemas que no admiten una descomposición que sea a la vez sin pérdida, FNBC, y preserve dependencias. Por ejemplo, consideremos

Dir (Barrio, Ciudad, Código_Postal)

y supongamos que un código postal pertenece a una única ciudad, y un barrio de una ciudad tiene un único código postal, o sea:

Código_postal → Ciudad
Barrio Ciudad → Código_postal

Supongamos además que dos ciudades distintas pueden tener un barrio del mismo nombre, o sea, la df *Barrio → Ciudad* no se verifica. Existen dos claves:

(Código_postal, Barrio), (Barrio, Ciudad)

La dependencia

Código_postal → Ciudad viola FNBC; pero ninguna descomposición de este esquema va a preservar la dependencia *Barrio Ciudad → Código_postal*.

A continuación describiremos en más detalle el algoritmo bosquejado más arriba.

Algoritmo 5: Obtiene una descomposición sin pérdida de información donde cada esquema resultante está en FNBC.

Entrada: Un esquema relacional $R(A_1, \ldots, A_m)$ y un cubrimiento mínimal F_M para el conjunto de dfs F

Salida: Una descomposición $\rho = \{R_1, \ldots, R_k\}$ de R tal que $R_i \in \rho$ esté en FNBC con respecto a F_i, la proyección de F sobre R_i, $\forall i$

Procedimiento

$Z := R$ [Z será el esquema de la descomposición que posiblemente no esté en FNBC]

repeat
descomponer Z en $Z - A$ y XA, donde XA está en FNBC y $X \rightarrow A$;
$\rho := \rho \cup R_i\ (XA)$;
$Z := \rho \cup Z - A$;
until Z no pueda ser descompuesto;
$\rho := \rho \cup Z$
end

Procedimiento descomponer:
if Z no contiene ningún A y B tal que $A \in (Z - AB)^+$
return Z está en FNBC y no puede ser descompuesto
else
hallar A y B
$Y := Z - B$;
while Y contiene A y B tal que $(Y - AB)^+ \rightarrow A$
$Y := Y - B$;
return la descomposición $Z - A$ e Y [Y es XA en el procedimiento principal]
end

Este algoritmo tiene complejidad de tiempo polinomial: si hay m atributos, habrá m^2 pares de atributos AB, como el cálculo de la clausura de un conjunto de atributos es de orden lineal y la cantidad de veces que se realiza el cómputo del test $(Y - AB)^+ \rightarrow A$ está acotado por m^3, luego el tiempo total es evidentemente polinomial.

Por otra parte, se ha probado que es NP-completo decidir si un esquema dado cualquiera está en FNBC.

Tercera Forma Normal

Como no es siempre posible obtener una descomposición sin pérdida que preserve dependencias en FNBC, debemos considerar como alternativa una

forma normal algo menos exigente, llamada tercera forma normal (3FN). Un esquema de relación R está en 3FN si para toda dependencia funcional no trivial $X \rightarrow Y$, o bien X es una superclave de R, o Y es un subconjunto de alguna clave de R.

Es obvio que todo esquema que satisface FNBC también satisface 3FN. En cambio, lo converso no es cierto. Una dependencia que viola FNBC pero no 3FN se llama *dependencia transitiva*. Por ejemplo, volvamos al esquema

Dir (Barrio, Ciudad, Código_postal)

con las dependencias

Código_postal → *Ciudad*
Barrio Ciudad → *Código_postal*

Como el atributo *Ciudad* es un subconjunto de la clave *(Barrio, Ciudad)*, la dependencia *Código_postal* → *Ciudad* no es una violación de 3FN. Éste es un ejemplo de un esquema que satisface la 3FN, pero no FNBC. Esto nos demuestra que la tercera forma normal no es suficiente para eliminar totalmente la redundancia; la información de que cierto código pertenece a cierta ciudad se va a repetir para cada barrio de esa ciudad. Pero la ventaja es que existe un algoritmo que genera a partir de cualquier esquema una descomposición sin pérdida, que preserva dependencias, y en 3FN, lo cual no es posible para FNBC.

En lo que sigue formalizamos el algoritmo antes mencionado.

Algoritmo 6: Obtiene una descomposición sin pérdida de información, donde cada esquema resultante está en 3FN.

Entrada: Un esquema relacional $R(A_1, \ldots, A_m)$ y un cubrimiento minimal F_M para el conjunto de dfs F.

Salida: Una descomposición ρ de R tal que $R_i \in \rho$ esté en 3FN con respecto a F_i, la proyección de F sobre R_i, $\forall i$.

Primo es parte de una clave

Procedimiento

```
        if (F_M contiene X → A tal que X ∪ A = R)        {Caso especial}
            return = ρ R
        endif;
        ρ := ∅;
        i := 1;
        for (cada X → A en F_M)
        {agregar a la descomposición un esquema relacional sobre los
                atributos XA}
            ρ := ρ ∪ R_i(XA)
            i := i + 1
        endfor
{ρ está ahora en 3FN pero puede haber pérdida de información. Lo si-
guiente asegura que la descomposición sea sin pérdida}
        if (según Algoritmo 3 la descomposición tiene pérdida de infor-
                mación)
            Sea K cualquier clave de R
        ρ := ρ ∪ {R_i(K)}
        endif
    end
```

El Algoritmo 6 es lineal en p, donde p es la cantidad de df's en F_M, salvo por la verificación que se realiza al final con el Algoritmo 3. El tiempo de este último domina el tiempo de este algoritmo de normalización a 3FN.

Otras Formas Normales

Hemos mencionado que existen otras formas normales: la introducción de una tercera forma normal hace sospechar que existen la primera y la segunda. La primera forma normal simplemente establece que los componentes de cada tupla son valores atómicos y no conjuntos o tuplas. Actualmente esta restricción forma parte de la definición del modelo relacional, o sea que todos los esquemas que consideramos en este libro están por definición

en primera forma normal. La segunda forma normal es una forma más débil de la tercera, y es de interés puramente histórico.

Veamos ahora cómo se puede ir más allá de la forma normal de Boyce-Codd. Consideremos, por ejemplo, el esquema de relación

R(Anúmero, Materia, Carrera, Nota).

Supongamos ahora que un alumno puede estar inscripto en varias carreras, y que una misma materia puede ser común a varias carreras. Si un alumno está inscripto en las carreras de computación, matemática y física, y ha rendido tres veces examen de Entropía II, una materia común a las tres carreras, necesitamos 9 tuplas para representar esta información. O sea, la información sobre notas se repite en forma redundante para cada carrera. Observemos que no existe ninguna dependencia funcional no trivial en este esquema y por lo tanto satisface FNBC. Esto nos muestra que la FNBC tiene limitaciones en cuanto a su poder de evitar la redundancia. El problema es que si representamos en el mismo esquema de relación dos conexiones independientes entre sí, aunque ninguna de ellas sea funcional, tendremos redundancia. En este ejemplo, podríamos descomponer en

Cursa (Anúmero, Carrera)
Resultado (Anúmero, Materia, Nota)
Programa (Carrera, Materia).

y evitar así la redundancia. La teoría del diseño relacional se extiende a cubrir este tipo de situaciones mediante la noción de *dependencia multivaluada*, que analizaremos en la próxima sección, y que lleva a la definición de cuarta forma normal, y su generalización, llamada *dependencia de junta*, que lleva a su vez a la definición de la quinta forma normal o forma normal de proyección-junta.

5.4. DEPENDENCIAS MULTIVALUADAS

Un ejemplo servirá para poner en evidencia el problema planteado en la sección anterior.

Supongamos que tenemos un esquema llamado Profesional con atributos Nombre, Título e Idioma para una tabla

los profesionales de una empresa. Algunas personas pueden tener varios títulos y hablar varios idiomas. Una parte de esa tabla podría ser la siguiente:

Profesional:	Nombre	Título	Idioma
	Hugo	Agrimensor	Inglés
	Hugo	Ingeniero	Francés
	Hugo	Agrimensor	Francés
	Hugo	Ingeniero	Inglés
	Luis	Abogado	Italiano
	Luis	Abogado	Inglés

Se puede observar que, dado que el título no está vinculado con el idioma que la persona habla, debemos registrar en la tabla una fila por cada idioma y esto para cada título. En consecuencia, registramos dos veces que Hugo es Agrimensor, dos veces que es Ingeniero y también dos veces que habla Inglés y que habla Francés. Del mismo modo registramos dos veces que Luis es Abogado.

Evidentemente no existe en el caso planteado ninguna dependencia funcional, aparte de las triviales. En consecuencia la tabla está normalizada hasta FNBC. ¿Cómo explicamos entonces la redundancia? Tal como dijimos anteriormente, ésta surge de representar en el mismo esquema las dos conexiones Nombre-Título y Nombre-Idioma, independientes entre sí y que se verifican por la independencia entre los atributos Título e Idioma. Esta información semántica es capturada por la ya mencionada noción de dependencia multivaluada.

Una dependencia multivaluada expresa que dos conjuntos de atributos en un esquema son mutuamente independientes.

Si R es un esquema de relación y $XY \subseteq R$, podemos definir entonces una dependencia multivaluada (dmv) $X \rightarrow\rightarrow Y$ sobre R, que se lee X multideterminа Y, como una restricción aplicable a cualquier instancia r de R, según la cual para cada valor de X existe un conjunto de valores de Y y ese conjunto es independiente de los valores de los restantes atributos $R - XY$.

Más formalmente, la dmv $X\ Y$ se verifica en R, si para cualquier instancia r de R se cumple que si existen dos tuplas t_1 y t_2 en r para las cuales $t_1[X] = t_2[X]$, luego deben existir otras dos tuplas t_3 y t_4, también en r tales que:

i) $t_1[X] = t_2[X] = t_3[X] = t_4[X]$,
ii) $t_3[Y] = t_1[Y]$ y $t_3[R - XY] = t_2[R - XY]$,
iii) $t_4[Y] = t_2[Y]$ y $t_4[R - XY] = t_1[R - XY]$.

Esta definición de dmv nos dice que, intercambiando los valores que corresponden a los atributos de Y entre t_1 y t_2, obtendremos otras dos tuplas que necesariamente deben también estar en la relación r. Por ejemplo, en la relación *Profesional* definida más arriba vale la dmv *Nombre* $\rightarrow\rightarrow$ *Título*. Supongamos ahora que t_1 y t_2 son, respectivamente, la primera y segunda filas. Luego intercambiando entre ellas los valores para *Título*, en este caso *Agrimensor* e *Ingeniero*, nos produce precisamente las filas tercera y cuarta de la tabla.

Una forma de eliminar la redundancia en la relación *Profesional* es descomponerla en otras dos relaciones con atributos XY y XZ, con $Z = R - XY$. De esta forma nos queda

Prof1	Nombre	Título
	Hugo	Agrimensor
	Hugo	Ingeniero
	Luis	Abogado

Prof2	Nombre	Idioma
	Hugo	Inglés
	Hugo	Francés
	Luis	Italiano
	Luis	Inglés

Observemos que *Prof1* $|\times|$ *Prof2* da exactamente igual a *Profesional*. Esto ejemplifica la siguiente propiedad de las dmv:

Sea r una instancia del esquema de relación R y X, Y, Z subconjuntos de R, donde $Z = R - XY$, *la relación r satisface la dmv $X \rightarrow\rightarrow Y$ si y sólo si r descompone sin pérdida de información sobre los esquemas $R_1 = XY$ y $R_2 = XZ$.*

Otra observación es que las df's son un caso particular de las dmv's.

Axiomas y Reglas de Inferencia para Dependencias Multivaluadas

Afortunadamente existe un conjunto *válido* y *completo* de axiomas que nos permite, dado un conjunto inicial de df's y dmv's, *inferir* nuevas dependencias. Este conjunto está integrado por los Axiomas de Armstrong, tres

axiomas específicos para dmv's y dos axiomas que vinculan dfs con dmv's.

Los tres axiomas específicos son:

Sean $X, Y, Z, W \subseteq R$.

ADMV1. *Complementación*: De $X \twoheadrightarrow Y$ inferir $X \twoheadrightarrow R - XY$.

ADMV2. *Aumento*: Para cualquier W, si $V \subseteq W$, de $X \twoheadrightarrow Y$ inferir $XW \twoheadrightarrow YV$.

ADMV3. *Transitividad*: De $X \twoheadrightarrow Y$ e $Y \twoheadrightarrow Z$ inferir $X \twoheadrightarrow Z - Y$.

Los axiomas de conexión entre dfs y dmv's son los siguientes:

ADMV4. *Conversión*: De $X \rightarrow Y$ inferir $X \twoheadrightarrow Y$.

ADMV5. *Interacción*: De $X \twoheadrightarrow Y$ y si existe un W tal que $W \cap Y = \emptyset$, $W \rightarrow Z$ y $Z \subseteq Y$, inferir $X \rightarrow Z$.

EJEMPLO 5.8: consideremos el esquema $R(A, B, C, D, E)$ y sea $D = A \twoheadrightarrow BC$, $DE \twoheadrightarrow C$]. La siguiente es una derivación de $AE \twoheadrightarrow BD$ usando los axiomas:

1. $A \twoheadrightarrow BC$	(dada)
2. $A \twoheadrightarrow DE$	(complementación de 1)
3. $DE \twoheadrightarrow C$	(dada)
4. $A \twoheadrightarrow C$	(transitividad de 2, 3)
5. $AE \twoheadrightarrow C$	(aumento de 4 con $W = E$ y $V = \emptyset$)
6. $AE \twoheadrightarrow BD$	(complementación de 5)

De la misma manera que en el caso de las dfs, existen reglas de inferencia adicionales para las dmv's que también, obviamente, se derivan de los axiomas.

Reglas de Inferencia Adicionales para Dependencias Multivaluadas

R1. *Unión*: De $X \twoheadrightarrow Y$ y de $X \twoheadrightarrow Z$ inferir $X \twoheadrightarrow YZ$.

R2. *Pseudotransitividad*: De $X \twoheadrightarrow Y$ y de $YW \twoheadrightarrow Z$ inferir $XW \twoheadrightarrow Z - WY$.

R3. *Pseudotransitividad Mixta*: De $X \twoheadrightarrow Y$ y de $XY \rightarrow Z$ inferir $X \rightarrow Z - Y$.

R4. *Descomposición*: De $X \twoheadrightarrow Y$ y de $X \twoheadrightarrow Z$ inferir:
$X \twoheadrightarrow (Y \cap Z)$,
$X \twoheadrightarrow (Y - Z)$ y
$X \twoheadrightarrow (Z - Y)$.

En el punto 5.2 vimos el concepto de clausura de un conjunto de atributos. Éste nos permitía determinar, dado F y una nueva df $X \rightarrow Y$, si la nueva df pertenecía o no a F^+. ¿Existirá algo similar para las dmv's? En el título siguiente analizamos un algoritmo que permite resolver el problema de la implicación para dmv's.

Base de Dependencias

El problema de implicación para dmv's se puede plantear de la forma siguiente: dado un conjunto M de dmv's y una dmv m, ¿es posible derivar m desde M utilizando los axiomas ADMV (y las reglas)?

Sea $C = (S_1, \ldots, S_n)$ una colección de conjuntos de atributos no vacíos, tal que resulta cerrado bajo las operaciones de unión, intersección y diferencia y $U = S_1 \cup S_2 \cup \ldots \cup S_n$. Luego C contiene una subcolección única $D = (T_1, \ldots, T_g)$ de conjuntos no vacíos y disjuntos dos a dos, tal que todo miembro de C resulta de la unión de componentes seleccionados de D. D se denomina una *base* para C. D es *minimal* si es la base con el mínimo número de componentes. D es una partición para U.

Dado un [illegible] y todos los Y_i que figuran como lados derechos de dmv's $X \twoheadrightarrow Y_i$, $X \twoheadrightarrow Y_j$ [illegible] de M^+, por las reglas de Unión y de Descomposición para dependencias multivaluadas, la familia de los Y_i resulta cerrada bajo las operaciones de unión, intersección y diferencia. La base para la colección de los Y_i se denomina la *base de dependencias* de X y se indica por $BDep(X)$. Una dmv $X \twoheadrightarrow Y \in M^+$ si y sólo si Y es la unión de componentes de $BDep(X)$.

El siguiente algoritmo permite calcular las bases de dependencias para un conjunto de atributos $X \subseteq R$, dado un conjunto de dependencias multivaluadas M.

Algoritmo 7: Cálculo de las bases de dependencias $BDep(X)$.

Entrada: Un esquema R y un conjunto de dmv's M. Las dependencias funcionales de R, en virtud del ADMV4 (Conversión), se consideran como dependencias multivaluadas.

Salida: La base de dependencias de X con respecto a M.

Procedimiento:

$T := R - X$;
while haya cambios a T do Inicialmente se prueba $V = T$
  if existe $V \in T$ e $Y \rightarrow\rightarrow Z \in M$ tal que $V \cap Y = \emptyset$ y $V \cap Z \neq \emptyset$
    then reemplazar V con $V \cap Z$ y $V - Z$;
end (while) $T \cup V$
BDep(X) := $T \cup \{A \mid A \in X\}$;
end.

EJEMPLO 5.9: sea $R(A, B, C, D, E, I)$, $M = \{A \rightarrow\rightarrow EI, C \rightarrow\rightarrow AB\}$ y queremos hallar $BDep(AC)$. Siguiendo el algoritmo definimos $T = BDEI$. Inicialmente el único V es igual a T. Seleccionamos la dependencia $A \rightarrow\rightarrow EI$ y reemplazamos $V = BDEI$ por $V \cap Z = EI$ y $V - Z = BD$. En la siguiente pasada consideramos $V = BD$ y ahora seleccionamos la dependencia $C \rightarrow\rightarrow AB$, por lo que debemos reemplazar $V = BD$ por B y D. En este momento resulta $T = \{B, D, EI\}$ y no podemos hacer más cambios. Luego $BDep(X) = \{A, B, C, D, EI\}$.

La complejidad de tiempo del Algoritmo 7 está acotada por un polinomio que depende del tamaño de M y de R.

Cuarta Forma Normal (4FN)

Antes de definir 4FN, introduciremos los conceptos de *dmv trivial* y de descomposición sin pérdida en presencia de dmv's.

Decimos que, dados R y $X, Y \subseteq R$, una dmv $X \rightarrow\rightarrow Y$ es trivial si:

- $R = XY$

o

- $Y \subseteq X$, incluyendo el caso $Y = \emptyset$.

Dado un esquema de relación R y un conjunto de dfs y dmv's M asociado, una descomposición de R en $\rho = \{R_1, R_2\}$ es sin pérdida si y sólo si

$$(R_1 \cap R_2) \rightarrow\rightarrow (R_1 - R_2) \in M^+ \text{ o}$$
$$(R_1 \cap R_2) \rightarrow\rightarrow (R_2 - R_1) \in M^+.$$

Como puede observarse, el resultado anterior sirve como un test sólo para el caso en que la descomposición de R se haga en dos esquemas R_1 y R_2. Si los esquemas son más de dos, se aplica una generalización del algoritmo dado para dependencias funcionales y que veremos más adelante.

Definimos ahora 4FN: Un esquema R está en 4FN con respecto a un conjunto M de dependencias funcionales y multivaluadas si para toda dmv no trivial, de la forma $X \rightarrow\rightarrow Y$, X es una superclave para R.

Observamos que cualquier relación que satisface 4FN automáticamente satisface FNBC y también, obviamente, 3FN. Para ver que 4FN implica FNBC sigamos el siguiente razonamiento: supongamos que R satisface 4FN y $X \rightarrow A$ viola FNBC. Luego $A \in X$ y X no es superclave de R. Por lo dicho, debe existir al menos un atributo B en R, tal que $B \in R - AX$. Puesto que de $X \rightarrow A$ se infiere $X \rightarrow\rightarrow A$ (Axioma de Conversión) y esa dmv es no trivial, resulta que R no está en 4FN (Contradicción). Así concluimos que R también satisface FNBC.

Otra observación es que cualquier dmv no trivial que no viola 4FN en R es realmente una df, ya que el lado izquierdo es una superclave de R.

Dado un esquema R y un conjunto de dfs y dmvs M, el siguiente algoritmo permite hallar una descomposición $\rho = \{R_1, \ldots, R_k\}$, sin pérdida de información y tal que cada R_i satisface 4FN.

Algoritmo 8: Obtiene una descomposición sin pérdida de información, donde cada esquema resultante está en 4FN.

Entrada: Un esquema relacional $R(A_1, \ldots, A_m)$ y un conjunto de dfs y mvd's M.

Salida: Una descomposición $\rho = \{R_1, \ldots, R_k\}$ de R tal que $R_i \in \rho$ esté en 4FN con respecto a M_i, la proyección de M sobre R_i, $\forall i$.

Procedimiento

```
Z := R; [Z será el esquema de la descomposición que posiblemente
            no esté en 4FN]
repeat
    if existe una dmv X →→ Y que viola 4FN en Z then
      begin
          descomponer Z en Z - Y y XY, donde XY está en 4FN;
          ρ := ρ ∪ R_i (XY);
          Z := Z - Y;
      end
until Z no pueda ser descompuesto;
ρ := ρ ∪ Z
end.
```

Observamos que en la dmv $X \rightarrow\rightarrow Y$ del Algoritmo 8 podemos considerar, sin pérdida de generalidad, que $X \cap Y = \emptyset$, puesto que si vale $X \rightarrow\rightarrow Y$, entonces también vale $X \rightarrow\rightarrow (Y-X)$ [De $X \rightarrow X$ se infiere $X \rightarrow\rightarrow X$ (Axioma de Conversión) y luego aplicamos la regla de descomposición a $X \rightarrow\rightarrow Y$ y $X \rightarrow\rightarrow X$].

Un aspecto importante en este algoritmo es la consideración de las dependencias que resultan de la proyección de M^+ sobre Z, en cada paso.

El siguiente es un ejemplo de aplicación del algoritmo 8 tomado de la bibliografía (Zaniolo-Melkanoff [212]).

EJEMPLO 5.9: Los atributos de interés para un organismo como la Dirección de Tránsito son:

A *Lic*	número de licencia del motor de vehículos
B *Fábrica*	fabricante de motor de vehículos
C *Modelo*	modelo de vehículo
D *Año*	año de fabricación del vehículo
E *Valor*	valor actual del vehículo
F *Prop*	identificador único de una persona
G *CondLic*	número de licencia de conductor
H *Viol*	código numérico para violación de tránsito
I *Fecha*	fecha de violación

Las dependencias son:

A B D C E F G

1. *Lic* → *Fábrica Año Modelo Valor Prop CondLic*
2. *Lic* →→ *Viol Fecha*
3. *Fábrica Año Modelo* → *Valor*
4. *Fábrica Año Modelo* →→ *Lic Prop CondLic Viol Fecha*
5. *Prop* → *CondLic*
6. *CondLic* → *Prop*
7. *Prop* →→ *Lic Fábrica Año Modelo Valor* | *Viol Fecha*
8. *CondLic* →→ *Lic Fábrica Año Modelo Valor* | *Viol Fecha*

De considerar la dependencia 5 obtenemos el esquema [*Prop, CondLic*] sobre el que se proyectan 5 y 6. De 7 resulta [*Prop, Viol, Fecha*] con dependencia *Prop* →→ *Viol Fecha*. A este momento, en Z ya hemos eliminado *CondLic*, *Viol* y *Fecha*. De 3 obtenemos [*Fábrica, Año, Modelo, Valor*] con la dependencia 3.

Finalmente, al considerar la dependencia 1 nos queda

[*Lic, Fábrica, Año, Modelo, Prop*]

y la dependencia es *Lic* → *Fábrica Año Modelo Prop*.

Dependencias Multivaluadas Embebidas

Es posible que, dada una relación *r* sobre un esquema *R* que no está en 4FN, al descomponerla en el proceso de normalización, ahora satisfaga una nueva dmv que originalmente no existía como tal. Denominamos dmv embebida a una dmv que se satisface en una proyección de una relación, aunque no necesariamente en la relación misma.

Una dependencia multivaluada embebida(edmv) es una sentencia de la forma $X \rightarrow\rightarrow Y \mid Z$, donde $Y \cap Z \subseteq X$. La edmv $X \rightarrow\rightarrow Y \mid Z$ se satisface en una relación *r* sobre *R*, donde $XYZ \subseteq R$, si la dmv $X \rightarrow\rightarrow Y$ se satisface en $\pi_{XUYUZ}(r)$.

EJEMPLO 5.10: Supongamos que tenemos un esquema *R*(*A, P, E, O*), donde *A* es Artesano, *P* representa a aPrendices que trabajan con un artesano, *E*

es Especialidad del artesano y O el añO en el que el aprendiz aprendió la especilidad E. Las restricciones aplicables a este esquema son: cada artesano tiene varios aprendices, un artesano tiene varias especialidades y un aprendiz aprende una especialidad en un año dado. La única dependencia válida en R es $PE \rightarrow O$. Analicemos si vale la dmv $A \rightarrow\rightarrow P \mid E$:

En una fracción de la tabla para R tenemos

R	A	P	E	O
	a1	p1	e1	o1
	a1	p2	e2	o2

utilizando la definición de dmv puedo generar otras dos nuevas filas

a1	p2	e1	o1
a1	p1	e2	o2

las que no necesariamente son válidas, ya que no es obligatorio que p2 aprenda la especialidad e1 en el mismo año o1 que p1.

Si aplicamos el proceso de normalización a 4FN considerando la df obtenemos $R_1(P, E, O)$ y $Z = APE$

Ahora sobre APE vale la dmv $A \rightarrow\rightarrow P$ y también $A \rightarrow\rightarrow E$, por el axioma de complementación, por lo que ambas son edmv's en $APEO$. Si continuamos con la normalización llegamos a una descomposición $R_1(PEO)$ $R_2(AP)$ y $R_3(AE)$

5.5. DEPENDENCIAS DE JUNTA

Una dependencia multivaluada es un caso particular de una clase más general de dependencia de datos denominada *dependencia de junta (dj)*.

Dado R podemos definir una dependencia de junta $|\times|[R_1, \ldots, R_k]$, donde $\{R_1, \ldots, R_k\}$ es una descomposición de R, como una restricción sobre R tal que cualquier instancia $r(R)$ es la junta sin pérdida de sus proyecciones sobre los R_i, es decir, $r = \pi_{R_1}(r) \mid\times\mid \ldots \mid\times\mid \pi_{R_k}(r)$. Del mismo modo, una dmv $X \rightarrow\rightarrow Y$ sobre R se puede expresar como la dj $|\times|[XY, X(R-XY)]$, puesto que existe un resultado de la teoría de las dmv's según el cual *r(R) satisface la dmv $X \rightarrow\rightarrow Y$ si y sólo si r descompone sin pérdida sobre los esquemas XY y $X(R-XY)$*

Una dj también se puede definir en términos similares a los de las dmv's,

es decir, por medio de restricciones a los valores corrientes de una tabla que, como en el caso de dmv's, definen la existencia de alguna otra tupla:

La dj $|\times|[R_1, \ldots, R_k]$ se satisface para R si restringe los valores de toda instancia $r(R)$ tal que, si existen k tuplas $t_1, \ldots, t_k$ que satisfacen $t_i[R_i \cap R_j] = t_j[R_i \cap R_j]$, luego también existe en r otra tupla t_{k+1} definida por $t_{k+1}[R_i] = t_i[R_i]$ para $1 \le i \le k$.

Previo a definir la quinta Forma Normal definimos Dependencia de Junta Trivial: una dj $|\times|[R_1, \ldots, R_k]$ sobre R es trivial si $R_i = R$ para algún $1 \le i \le k$.

Quinta Forma Normal (5FN)

Dado un esquema R y un conjunto de dfs y dj's D, R está en 5FN (o Forma Normal Proyección/Junta) con respecto a D, si para toda dj de D se cumple que

i. la dj es trivial

o

ii. la dj es implicada por las dfs que tienen a claves de R como li

Implicación de Dependencias

Dado un conjunto de dfs y dj's D, queremos saber si dada una nueva dj j_1, ésta es implicada por D, es decir, ¿$D \models j_1$? Para esto usaremos una generalización de la versión dada anteriormente del algoritmo Chase.

El tableau inicial se construye de la misma forma que antes, teniendo en cuenta que ahora la descomposición a testear es la dada por j_1. Si j_1 fuera, por ejemplo, $|\times|[S_1, S_2, S_3]$, luego construimos el tableau en base a la descomposición de R en S_1, S_2, S_3:

T:	A_1	A_2	...	A_m
S_1				
S_2				
S_3				

En esta versión más general, las transformaciones al tableau serán de dos tipos, a saber, por aplicación de reglas F (referidas a dfs) y por aplicación de reglas J (referidas a djs)

Sea R un esquema de relación y T un tableau:

a) Regla F: toda df de la forma $X \rightarrow A$ de D tiene asociada una regla F, la que puede ser aplicada a transformar T si éste contiene filas apropiadas. Si T contiene dos filas denominadas u y v, tales que $u[X] = v[X]$ y $u[A] \neq v[A]$, podemos aplicar la regla F $X \rightarrow A$ igualando $u[A]$ con $v[A]$ de la siguiente forma: si una de ellas es una variable distinguida, las dos filas son forzadas a contener la variable distinguida en la columna A; si ambas contienen variables no distinguidas forzamos a una de ellas (la de mayor subíndice) a contener la otra variable. Este cambio produce un nuevo tableau llamado T_1.

b) Regla J: dada la dj $Q \bowtie [Q_1, \ldots, Q_p]$ definida sobre R y $u_1, \ldots, u_p$ filas de T juntables sobre Q con resultado u, la aplicación de esta regla J a T produce un nuevo tableau $T_1 = T \cup u$.

El uso original del Chase fue verificar si una descomposición dada era sin pérdida de información, es decir, si $r = r_1 \bowtie \ldots \bowtie r_k$. Pero como vimos ésa es la definición de dj. Luego si probamos que la descomposición dada por una dj es sin pérdida con respecto a D, lo que estaremos probando es que la dj se satisface en R, sujeto a las restricciones de D, y por lo tanto D implica j. Como consecuencia, *D implica j si y sólo si el tableau final presenta una fila con solamente variables distinguidas.*

El algoritmo Chase tiene complejidad de tiempo exponencial: si el tableau T original tiene m columnas y k filas, el tableau final por aplicación de las reglas J, puede tener k^m filas.

EJERCICIOS

E1. Dado R (H, I, K, L, M, O) y el conjunto de dependencias funcionales
$F = \{H \rightarrow IO,\ O \rightarrow HO,\ KM \rightarrow L,\ L \rightarrow MK,\ M \rightarrow K,\ HK \rightarrow M\}$,
hallar todas las claves candidatas.

E2. Dado R (A, B, C, D, E) y el siguiente conjunto de dfs
$F = \{A \rightarrow BC,\ CD \rightarrow E,\ B \rightarrow D,\ E \rightarrow A\}$,
probar, utilizando los axiomas, que las siguientes dependencias pertenecen a F^+:

a) $E \rightarrow B$
b) $EA \rightarrow CA$
c) $AD \rightarrow E$
d) $AC \rightarrow E$

E3. Probar el axioma de transitividad mediante la definición de df.

E4. Probar, utilizando los axiomas para dfs las siguientes reglas:

a) De $X \rightarrow Y$, $X \subseteq V$, YZ inferir $V \rightarrow Z$.
b) De $X \rightarrow Y$, $W \subseteq Y$, $W \rightarrow Z$ inferir $X \rightarrow Z$.
c) De $X \rightarrow Y$, $X \subseteq V$, $W \subseteq Y$, $W \rightarrow Z$ inferir $V \rightarrow Z$.

E5. Dado $R(A, B, C, D, E, F, G)$ y el siguiente conjunto de dfs:
$F = \{AB \rightarrow C,\ C \rightarrow A,\ BC \rightarrow D,\ ACD \rightarrow B,\ D \rightarrow EG,\ BE \rightarrow C,\ CG \rightarrow BD,\ CE \rightarrow AG\}$,
hallar un cubrimiento minimal F_M.

E6. Sea R un esquema relacional, F un conjunto de dfs y F_M un cubrimiento minimal de F. Probar que:

a) Si $X \rightarrow A \in F_M$, $X \subseteq R$, $A \in R$, no puede ser X superclave de R sin ser, a su vez, clave de dicho esquema.
b) Si $X \rightarrow A \in F$, $X \subseteq R$, $A \in R$, y $F \models X_1 \rightarrow X_2$, con $X_1 \subset X$, $X_2 \subset X$, $X_2 \not\subset X_1$, entonces $X \rightarrow A \notin F_M$.
c) Si $F \models X \rightarrow Z$, $X \subseteq R$, $Z \subseteq R$, $Z \cap X = \emptyset$, entonces para todo atributo $A_i \in Z$ hay un $X_i \subseteq R$, tal que $X_i \rightarrow A_i \in F_M$.
d) Si $C = \{A \mid \neg\exists X \subseteq R$ que cumpla que $X \rightarrow A \in F_M\}$, entonces cualquier clave K de R cumple con que $C \subseteq K$.

E7. Sea el esquema de relación $R(A, B, C, D, E, F, G, H)$ y el siguiente conjunto de dfs
$F = \{ABC \rightarrow E,\ FD \rightarrow A,\ AG \rightarrow E,\ D \rightarrow C,\ BC \rightarrow F,\ A \rightarrow H,\ F \rightarrow D,\ H \rightarrow G\}$.

a) Encontrar un cubrimiento minimal para F.
b) Encontrar todas las claves de R.
c) Mostrar que lo probado en el ejercicio E6 vale en este ejemplo.

E8. Dado el esquema de relación $R(A, B, C, D, E)$ y las siguientes instancias

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_2	c_2	d_1	e_1
a_2	b_1	c_2	d_1	e_1

A	B	C	D	E
a_1	b_2	c_1	d_2	e_1
a_2	b_2	c_1	d_2	e_2
a_2	b_2	c_2	d_2	e_2

A	B	C	D	E
a_1	b_1	c_2	d_2	e_2
a_2	b_1	c_2	d_2	e_2
a_1	b_2	c_1	d_1	e_1

a) Analizar si cada instancia cumple con las siguientes dependencias:
   i. $A \rightarrow BC$
   ii. $D \rightarrow E$
   iii. $\emptyset \rightarrow D$
   iv. $BC \rightarrow D$
b) Analice para cada caso las anomalías presentes en R tomando como conjunto de dependencias al formado por las dependencias de i a iv del punto anterior que se hayan verificado sobre la instancia.

E9. Sea $R(A, B, C, D, E, F, G, H, I)$ y el conjunto de dfs
$F = \{A \rightarrow B, CD \rightarrow F, H \rightarrow AD, I \rightarrow C, D \rightarrow H\}$
a) Decir si la descomposición $R_1(A, B, D)$ $R_2(D, E, F)$, $R_3(F, G, C)$, $R_4(C, H, I)$ es sin pérdida de información. Justificar.
b) Ídem para $R_1(A, B, C, D)$, $R_2(E, F, G)$, $R_3(H, I)$.

E10. Dado el siguiente tableau

	A	B	C	D	E
R_1	a_1	b_{12}	b_{13}	a_4	b_{15}
R_2	b_{21}	a_2	a_3	b_{24}	a_5
R_3	a_1	a_2	b_{33}	b_{34}	a_5

y un conjunto de dfs F. Si $R_1 = AD$, $R_2 = BCE$ y $R_3 = AE$, se pide:
a) Indicar cuáles dependencias harían falta en F de modo que la transformación del tableau inicial alcance este tableau (aunque éste no sea el tableau final)
b) Ídem anterior, pero que además garantice que al terminar el Chase se pruebe que la descomposición es sin pérdida de información

E11. Dado un esquema $R(A, B, C, D, E, F)$ y una descomposición $\rho = (R_1, R_2, R_3)$, con $R_1 = AC$, $R_2 = BCD$ y $R_3 = BEF$. Si asumimos que sobre cualquier instancia de R deben verificarse las dfs
$F = \{A \rightarrow DE, D \rightarrow C, BC \rightarrow D, B \rightarrow AF\}$

a) Encuentre una instancia I y una tupla u de R que cumplan con
   i. I satisface las dfs en F.
   ii. $\pi_{R_i}(I \cup \{u\})$ satisface las dfs en $\pi_{R_i}(F)$, para $1 \leq i \leq 3$.
   iii. $I \cup \{u\}$ no satisface las dfs en F.

E12. Sean el esquema de relación $R(A, B, C, D, E, F, G, H)$ y el siguiente conjunto de dfs
$F = \{ABC \rightarrow E, FD \rightarrow A, AG \rightarrow E, D \rightarrow C, BC \rightarrow F, A \rightarrow H, F \rightarrow D, H \rightarrow G\}$
a) Hallar un cubrimiento minimal para F.
b) Analizar si el esquema BCF admite alguna descomposición en FNBC, sin pérdida de información y preservando las dfs.
c) Normalizar R.

E13. Dado el esquema $R(A, B, C, D, E, G)$ y
$F = \{CE \rightarrow D, G \rightarrow B, D \rightarrow E, G \rightarrow D, D \rightarrow G\}$,
a) ¿Se encuentra R en 3FN? Si la respuesta es No, normalizar a 3FN.
b) Hallar una descomposición de R en FNBC. Verificar que no haya pérdida de información. Ídem pérdida de dfs. Si se pierden dfs, hallar otra descomposición en la que se pierdan otras dependencias.

E14. Sea el esquema de relación $R(P, F, U, I, O, C, D)$, donde P es el código de un préstamo en una biblioteca, F es la fecha del préstamo, U es el usuario lector a quien le fue concedido el préstamo, I es el código de inventario del objeto prestado, O es el nombre identificatorio de la obra involucrada, C es el número de ejemplar de la obra (puede haber más de un ejemplar de una misma obra), y D es la fecha pactada de devolución.
a) Proponer un conjunto de dfs que se verifiquen en el problema. Justificar dicha elección.
b) Determinar las claves de R.
c) Descomponer sin pérdida de información en esquemas en 3FN, definiendo claves y garantizando la preservación de todas las dfs.
d) ¿Es posible obtener lo mismo que en c) pero con todos los esquemas en FNBC? Justifique.

E15. Dados el esquema y las instancias del ejercicio E8:

a) Analizar si cada instancia cumple con las siguientes dependencias:

i. $A \rightarrow\rightarrow BC$

ii. $D \rightarrow E$

iii. $C \rightarrow D$

iv. $BC \rightarrow\rightarrow D$

b) Encontrar para los ítems i. a iv. del punto a) anterior, en caso que exista, una instancia que incluya a la instancia analizada y que cumpla con la dependencia en cuestión.

c) Mostrar las anomalías presentes en R bajo las restricciones establecidas en i. a iv.

E16. Dados el esquema de relación $R(A, B, C, D, E, F, G)$ el conjunto de df's y dmv's $M = \{A \rightarrow\rightarrow BCDF, A \rightarrow\rightarrow CDE, E \rightarrow B, BF \rightarrow\rightarrow C\}$, calcular las bases de dependencias para AG.

E17. Probar que

si $X \rightarrow\rightarrow Y$, $Y \rightarrow\rightarrow Z$ y $Z \rightarrow\rightarrow W$, luego $X(Y \cap Z) \rightarrow\rightarrow (W-Z)$

a) dando una instancia de una relación donde aparezcan las tuplas requeridas

b) utilizando los axiomas(y las reglas) para dmv's

E18. Sean $R(A, B, C, D, E, F)$ y $F = \{AE \rightarrow C, D \rightarrow F, C \rightarrow E, AB \rightarrow D\}$

a) Calcular las claves candidatas.

b) ¿En qué forma normal se encuentra el esquema?

c) Agregar $B \not\rightarrow\rightarrow C$ y responder nuevamente a) y b).

d) Descomponer en 4FN diciendo qué df's se perdieron.

E19. Sean $R(A, B, C, D, E, F, G)$ y

$$M = \{AB \rightarrow\rightarrow CF, A \rightarrow D, CD \rightarrow\rightarrow FG, DE \rightarrow E\}$$

a) Calcular las claves candidatas.

b) Agregar $F \rightarrow\rightarrow E$ y $D \rightarrow B$ y recalcular a)

c) ¿En qué forma normal se encuentran los subesquemas ABC, $ABDE$ y $ACFG$?

d) Descomponer en 4FN sin pérdida de información

E20. Sea el esquema de relación $R(E, D, S, G)$, donde E identifica a empleado, D al departamento donde trabaja, S es el sueldo del empleado y G el gerente del departamento.

Un empleado tiene un único sueldo, un departamento puede tener varios gerentes y un empleado puede trabajar en varios departamentos.

a) representar las restricciones enunciadas por medio de un conjunto de df's y dmv's. Justificar.

b) Normalizar R a 4FN.

E21. La siguiente es una vista parcial de una relación $r(R)$

R:	A	B	C
	1	2	3
	4	5	6
	4	2	7
	1	8	7

Si para esta relación se satisface la dj $|\times|[AB, BC, AC]$, ¿qué otras tuplas debe contener la relación?

NOTAS BIBLIOGRÁFICAS

Para una discusión detallada y rigurosa de los temas presentados en este capítulo, así como otros temas de teoría relacional, ver los libros de D. Maier [152], de J. Ullman [202] y, el más reciente, de Abiteboul, Hull, Vianu [2]; y también, para una visión general, los artículos de Beeri, Bernstein, Goodman [22] y Paris Kanellakis [132].

Un texto que combina el uso del modelo Entidad-Interrelación, a nivel conceptual, con un enfoque riguroso del diseño relacional es el de Mannila y Raiha [157]. Otros textos de interés son S. Alagic [8], Atzeni y De Antonellis [13], Elmasri y Navathe [80], Date [70] y R. Ramakrishnan y J. Gehrke [174].

Las df's y las primeras formas normales hasta la tercera, fueron definidas por Edgard Codd en Codd [61] y Codd [62], mientras que la FNBC se introdujo en Codd [63]. Los axiomas para df's fueron definidos en W. W. Armstrong [9]. El algoritmo citado para el cómputo de X^+ en tiempo proporcional a n se publicó en Beeri, Bernstein [21] mientras que el cómputo de un cubrimiento mínimal para F fue analizado en D. Maier [151].

El algoritmo para testear la preservación de dependencias funcionales en una descomposición se introdujo en Beeri, Honeyman [25].

El estudio de la descomposición de relaciones y sus esquemas se publicó en Rissanen [177] y otros como Maier, Mendelzon, Sadri, Ullman [153], mientras que el problema de equivalencia fue resuelto en Beeri, Mendelzon, Sagiv, Ullman [26].

Las dmv's fueron descubiertas en forma independiente por C. Delobel [73], R. Fagin [82] y C. Zaniolo [216]. La 4FN y 5FN fueron definidas por Ron Fagin en Fagin [82] y Fagin [84], respectivamente.

El algoritmo para 3FN fue publicado en P. Bernstein [33], mientras que el correspondiente a FNBC apareció en D. Isou, P. Fischer [201]. El algoritmo para descomposición a 4FN es de R. Fagin [82] mientras que el algoritmo 7 para el cálculo de las bases de dependencias de un conjunto de atributos fue introducido por C. Beeri [20].

Las dependencias multivaluadas embebidas aparecen también en Fagin [83], mientras que las dependencias de junta fueron definidas en Rissanen [179].

El algoritmo Chase aplicado a la implicación de dependencias es de D. Maier, A. Mendelzon, Y. Sagiv [154].

CAPÍTULO 6

ESTRUCTURAS FÍSICAS DE DATOS

Hasta ahora, hemos observado un sistema de base de datos "desde afuera", discutiendo cómo diseñar una base de datos, cómo formular consultas, etc., pero sin discutir la implementación interna del sistema. A partir de este capítulo nos acercamos más al punto de vista del implementador de un sistema de base de datos; en particular, en este capítulo daremos una breve síntesis de los métodos más comunes de organización de archivos para soportar bases de datos relacionales. El tema da para mucho más, pero como pertenece más bien al área de estructuras de datos que a la de bases de datos en sí, nuestra exploración sólo será superficial.

6.1. ARCHIVOS, REGISTROS, BLOQUES

Aunque el costo de la memoria principal ha decrecido exponencialmente sin interrupción desde las primeras memorias de núcleos magnéticos, paralelamente ha crecido la cantidad de información que se desea manejar. En consecuencia, los sistemas de bases de datos parten de la premisa de que la base de datos deberá residir en dispositivos de memoria secundaria, generalmente discos magnéticos. En memoria principal sólo será posible almacenar de una vez cierta fracción del total de datos, para lo cual el sistema maneja un conjunto de *buffers*.

Señalemos como digresión que la tecnología de memoria ha llegado a un punto donde es razonable para bases de datos de tamaño no demasiado grande, digamos del orden de varios megabytes, almacenarlas completamente en

memoria principal. La adaptación de técnicas de archivo tradicionales a es-
ta nueva situación es un tema de investigación activo hoy en día, y existen
sistemas, tales como *IMS Fast Path*, que ofrecen la opción de cargar una ba-
se de datos completamente en memoria principal. En el resto del capítulo ig-
noraremos esta posibilidad.

Vamos a suponer que el lector conoce la organización de una típica uni-
dad de disco. Para nuestros propósitos es suficiente suponer que el sistema
de manejo de archivos provee la abstracción de un *bloque de disco*, una se-
cuencia contigua de bytes identificada en forma única por una dirección de
bloque, y que se puede leer o modificar en forma eficiente. Típicamente el
tamaño de un bloque irá de los 128 bytes en máquinas muy pequeñas hasta
varios miles de bytes en las más grandes.

Desde el punto de vista del sistema de base de datos, las unidades de dis-
co tienen dos características fundamentales que las distinguen de la memo-
ria principal. La primera es la velocidad de acceso; dada la necesidad de mo-
ver mecánicamente un brazo de lectura/escritura, el acceso a un bloque
arbitrario de disco puede causar demoras del orden de milisegundos, varios
órdenes de magnitud mayores que las causadas por el acceso a memoria
principal. Esto implica que, una vez que el brazo se posiciona en el lugar de-
seado, es conveniente acceder una cantidad relativamente grande de bytes.
Dada esta relación entre los costos de acceso a memoria secundaria y a me-
moria principal, en las estimaciones de costo el factor dominante es casi
siempre la cantidad de bloques de disco que deben ser accedidos, y el tiem-
po de procesador utilizado es generalmente de mucho menor importancia.

La segunda característica fundamental de la memoria secundaria es que es
estable, o sea, la información almacenada en ella sobrevive caídas del sistema o
cortes de electricidad, salvo casos catastróficos. La memoria principal en cam-
bio es generalmente dinámica, o sea, la información almacenada en ella se pier-
de ante una caída de tensión, y, aunque no se pierda físicamente, es casi impo-
sible de recuperar después de una falla de software o hardware. Por lo tanto
para garantizar que la base de datos sobrevivirá en estado aceptable una caída
de cualquier origen, el sistema debe cuidar que toda información mantenida en
memoria principal sea de algún modo transferida a memoria estable a interva-
los frecuentes. Por otro lado, el manejo de transacciones requiere que una mo-
dificación a un bloque no se haga efectiva hasta tanto no se esté seguro que esa
transacción va a ser completada en su totalidad.

La mayor unidad de información manejada por el sistema de archivos es
por supuesto el *archivo*, que consiste en una secuencia lógica de *registros*. El
tamaño de un registro depende de la aplicación; si todos los registros de un

archivo tienen el mismo tamaño, hablamos de un archivo de registros de lon-
gitud fija, y en caso contrario, de longitud variable.

El primer problema a solucionar es cómo ubicar estos registros de longi-
tud arbitraria, sea fija o variable, en bloques de capacidad fija. Éste es un
problema clásico de estructuras de datos, que no discutiremos. Su solución
óptima depende de los costos que se quieran minimizar: utilización del es-
pacio en disco disponible, tiempo de lectura, tiempo de actualización, sim-
plicidad de implementación.

6.2. MANEJO DE BUFFERS

Como no es posible mantener en memoria principal todos los bloques de
disco que componen la base de datos, el sistema tiene que decidir cómo usar
el espacio disponible en forma de minimizar la cantidad de accesos a disco
necesarios para procesar una consulta o actualización. Llamamos *área de*
buffers o *memoria cache* a la zona de memoria principal usada para mante-
ner copias de bloques de disco. El *gestor de buffers* o *gestor de cache* es el mó-
dulo del sistema de base de datos que se encarga de administrar esta área.

Supongamos que otro módulo del sistema necesita leer cierto bloque de un
archivo. El gestor de buffers intercepta el pedido de bloque emitido por este otro
módulo. Si ese bloque ya se encuentra en un buffer de memoria principal, el
módulo que hizo el pedido recibe como respuesta la dirección de memoria del
buffer. Si no, el gestor de buffers hace que el bloque sea leído a un nuevo buf-
fer y devuelve la dirección de este buffer al módulo cliente. O sea, el gestor de
buffers es transparente a los módulos que requieren bloques.

Los lectores que conocen sistemas operativos de memoria virtual ya ha-
brán notado el problema que el gestor de buffers encuentra cuando debe leer
un nuevo bloque de disco pero todo el espacio de buffers se halla ocupado
por otros bloques. En este caso, alguno de esos bloques debe ser descartado
para poder reusar su buffer. El problema de elegir un bloque a descartar se
llama el *problema de reemplazo*. En un sistema operativo, una política de
reemplazo común es la llamada *LRU (Least Recently Used)*, que consiste en
reemplazar aquel bloque que fue usado menos recientemente. Esta política
no siempre es adecuada para sistemas de bases de datos.

Por ejemplo, supongamos que queremos calcular la junta de dos relacio-
nes R y S,

R |X| S

El método a usar será: leeremos secuencialmente todos los bloques de *R*. Para cada bloque de *R* leído, leeremos secuencialmente todos los bloques de S para determinar qué tuplas de S participan en la junta con tuplas del bloque actual de *R*. Observemos que, una vez leído y procesado completamente un bloque de R, ya no lo necesitamos. Por lo tanto, el gestor de buffers debería liberar inmediatamente el espacio usado por un bloque de *R* una vez que éste ha sido procesado, mientras que la estrategia *LRU* no lo haría hasta que muchos otros bloques sean usados y hagan que este bloque se convierta en el menos recientemente usado.

Consideremos en cambio los bloques de S. Una vez que se ha procesado un bloque de S, no volveremos a usarlo hasta tanto todos los otros bloques de S hayan sido procesados una vez más. O sea, el bloque de *S* usado *más recientemente* será el último en ser reusado. Esto implica que la estrategia óptima es exactamente la opuesta a la *LRU*.

La moraleja de este ejemplo es que el problema de reemplazo de buffers es más complejo en el caso de sistemas de bases de datos que en sistemas operativos, y, si el gestor de buffers cuenta con suficiente información de alto nivel sobre qué está intentando hacer el sistema en su conjunto, puede seleccionar estrategias que resulten en mayor eficiencia. Otro punto que no hemos considerado es que en un sistema multiusuario se estarán ejecutando múltiples operaciones en paralelo, lo cual vuelve el problema de encontrar un manejo óptimo global aún más complicado. Finalmente, la necesidad de garantizar que el sistema puede recuperarse de una caída impone restricciones adicionales sobre cuándo un bloque que ha sido modificado en su buffer puede o debe ser reescrito a disco.

6.3. REPRESENTACIÓN DE RELACIONES

Veamos ahora cómo el concepto abstracto de una tabla o relación puede implementarse en términos de archivos. La idea más simple es la de representar cada relación por medio de un archivo. Generalmente las tuplas de una relación pueden representarse como registros de longitud fija sin demasiado desperdicio de espacio. Esta organización es frecuente en sistemas para máquinas relativamente pequeñas, donde las relaciones no serán demasiado grandes y es importante tener una implementación simple que permita que el código del sistema en sí no ocupe demasiado espacio.

En sistemas de mayor envergadura puede ocurrir que se combinen varias relaciones en un mismo archivo. Para ver las ventajas que esto puede traer, consideremos la consulta SQL que lista todos los pediatras junto con el nombre del hospital en el cual trabajan:

```
select Medico.nombre, Hospital.nombre
  from Medico,Hospital
    where espec="pedia"
      and Medico.hosp = Hospital.código
```

Supongamos que Hospital y Médico están representadas por dos archivos separados. Para cada tupla de Hospital, tendremos que recuperar todas las tuplas de Médico que tengan el mismo código de Hospital. Veremos más adelante cómo los métodos de indexación o hashing permitirán hacer esto sin necesidad de una búsqueda secuencial sobre Médico; pero de todos modos, cada una de estas tuplas de Médico potencialmente puede residir en un bloque distinto obligándonos a efectuar un acceso a disco por cada registro deseado.

Supongamos en cambio que usamos un solo archivo para almacenar ambas relaciones, de la siguiente forma: las tuplas de Hospital y Médico se hallan entrelazadas; una tupla de Hospital está seguida físicamente en el disco por todas las tuplas de Médico que tienen el mismo código que ese hospital. La figura 6.1 muestra un ejemplo. Con esta organización una vez que ubicamos una tupla de Hospital, toda la información sobre los médicos de ese hospital se puede acceder leyendo la mínima cantidad de bloques posible en forma secuencial. Este tipo de esquema se llama *agrupamiento (clustering)* porque las tuplas se agrupan según el valor de algunos de sus atributos; en este caso código de hospital.

Como suele ocurrir, hemos ganado en el tiempo de ejecución de cierto tipo de consultas a costa de una pérdida en otras. Por ejemplo, si queremos simplemente listar todos los hospitales:

```
select *
from Hospital;
```

22	Rivadavia	412	Psiq
22	453	Toriani	Card
22	398	De la Mata	
13	Central	502	Pedia
13	301	Navarro	Pedia
13	197	Bernao	
45	De Niños	845	Pedia
45	607	Erico	
18	Italiano	987	Neuro
18	583	Artime	

Figura 6.1
Ejemplo de Agrupamiento

tendremos que acceder muchos más bloques que si la relación Hospital estuviera almacenada en un archivo separado. Esto sugiere que la decisión de cómo y cuándo usar agrupamientos no es trivial y depende de las frecuencias relativas de distintos tipos de consultas.

Diccionario de Datos

Para representar una base de datos relacional, necesitamos almacenar no sólo las relaciones mismas, sino también una cantidad de información adicional *acerca* de las relaciones. El conjunto de tal información se llama el *diccionario de datos*, o a veces la *metainformación*. Por ejemplo, el diccionario de datos debe contener datos tales como:

- nombre de las relaciones en la base de datos
- nombre de los atributos de cada relación
- dominio de cada atributo
- tamaño de cada relación
- método de almacenamiento de cada relación (por ejemplo, uso o no de agrupamiento)
- claves y otras ...
- nombres y definiciones de vi...

Además, deberá contener información sobre los usuarios y los poderes conferidos a éstos, por ejemplo:

- nombre de los usuarios
- costo del uso efectuado por cada usuario
- nivel de privilegio de cada usuario.

Algunos sistemas usan estructuras de datos especiales para almacenar y administrar toda esta información. Una solución más elegante consiste en notar que el diccionario de datos no es más que una base de datos acerca de la base de datos, y puede almacenarse y manejarse con los mismos métodos. Así podemos definir una relación que almacena los nombres y descripciones de cada relación (incluida ella misma), una relación que describe los atributos etc.

6.4. INDEXACIÓN

Una de las operaciones fundamentales en cualquier base de datos es el acceso a una tupla o registro dado el valor de su clave; por ejemplo, extraer el registro de un empleado dado el número de empleado o el registro de un vehículo dada su matrícula. Obviamente tales operaciones siempre pueden implementarse mediante un barrido secuencial del archivo en cuestión, pero si nos interesa sólo acceder un registro o una pequeña fracción de todos los registros, una búsqueda secuencial será inaceptable. Por esto es necesario agregar a los métodos básicos de representación que discutimos mas arriba otras estructuras de datos cuya única función es acelerar las operaciones de consulta y/o actualización. La primera de tales estructuras que discutiremos se conoce como *índice*. Distinguiremos dos tipos de índices, según si el archivo a indexar esté almacenado en forma secuencial o no.

vo se considera demasiado ...
de accesos baja y esencialmente independiente del tamaño del archivo, o a lo sumo logarítmica en éste. En este caso, una búsqueda binaria nos daría una cantidad de accesos a bloque de sólo $\log_2 B$, pero la búsqueda binaria requiere una secuencialidad estricta que será costoso mantener cuando el archivo sea actualizado.

Proponemos en cambio la construcción de un archivo auxiliar llamado *índice*. El índice consiste en una secuencia de pares (K_i, b_i) donde K_i es un valor de la clave del archivo a indexar y b_i es una dirección de bloque, con la propiedad de que, para toda clave $K_i \le K < K_{i+1}$, si el registro con clave K está en el archivo, debe estar en el bloque b_i. Podemos tomar como K_i a la clave mínima del bloque b_i.

Supongamos ahora que queremos hallar el registro con clave K; basta ubicar en el índice un par de claves $K_i \le K < K_{i+1}$ para saber en qué bloque se halla el registro deseado y obtenerlo con un sólo acceso. Si el índice es suficientemente pequeño para residir en memoria principal, este acceso es el único costo de la búsqueda (ignorando tiempo de procesador). De lo contrario, el índice es a su vez un archivo en el cual debemos ubicar un registro. Si el tamaño del índice lo justifica, podemos recursivamente construir un índice del índice, y así sucesivamente. Esto es lo que se llama un sistema de indexación de nivel múltiple; en este caso, el costo de acceso a un registro del archivo será proporcional a la cantidad de niveles.

Es frecuente en los sistemas de indexación de nivel múltiple hacer que cada nivel coincida con un nivel de la jerarquía en que está estructurada una unidad de disco: sector, pista, cilindro, volumen. Así podemos tener al nivel más alto un índice de volumen, que nos dice cuál es el registro de clave mínima en cada volumen; para cada volumen, un índice de cilindro, que nos da esta información para cada cilindro, dentro de cada cilindro, un índice de pista, etc.

No es sorprendente que una organización secuencial es costosa de mantener en la presencia de altas, bajas y modificaciones. Supongamos que

encadenada de bloques de overflow ...
a bloque necesarios para ubicar un registro, de modo que la performance se deteriora a medida que la longitud de las listas de overflow aumenta y se hace necesario periódicamente reorganizar el archivo.

Notemos también que ciertas altas y bajas requieren actualizar no sólo el archivo, sino también el índice. Por ejemplo, si queremos insertar un registro con clave K donde $K_i < K < K_{i+1}$, y el bloque b_i no tiene espacio, pero el bloque b_{i+1} sí lo tiene, puede convenir almacenar el registro en el bloque b_{i+1} y reemplazar K_{i+1} por K en el índice. En caso de baja, la eliminación de un registro cuya clave aparece en el índice requiere actualizar el índice reemplazando esta clave por la nueva clave mínima del bloque.

Índices Densos

Un método que no requiere secuencialidad del archivo es construir lo que se llama *índice denso*. Se trata simplemente de una lista de pares K_i, b_i tales que el registro con clave b_i se encuentra en el bloque b_i. Almacenando el índice en forma secuencial podemos acceder cualquier registro a un costo que es un bloque más que el costo de acceder el índice. En general, el costo de almacenamiento de un índice denso será considerablemente mayor que el de un índice de archivo secuencial.

Árboles-B

Una elegante generalización de la idea de índices de nivel múltiple es lo que se llama un *árbol-B* (la *B* probablemente debida a Rudolf Bayer, uno de los inventores de esta estructura). El árbol-B es la más comúnmente usada de un conjunto de estructuras que tratan de obtener la eficiencia en la con-

sulta del indexado secuencial sin sufrir el deterioro que este último exhibe bajo actualizaciones.

Un árbol-B de orden n es un árbol con dos propiedades. En primer lugar todo nodo, excepto la raíz, tiene no menos de $n/2$ y no más de n hijos. La raíz puede tener entre 0 y n hijos. En segundo lugar, el árbol debe ser *balanceado*, o sea, todo camino de la raíz a un nodo terminal debe tener la misma longitud. La primera propiedad garantiza que no se desperdicia demasiado espacio, ya que cada nodo del árbol debe tener espacio para n punteros de los cuales al menos la mitad será usada. La segunda propiedad asegura, como veremos, que el tiempo de búsqueda será logarítmico en la cantidad de registros.

Describamos primero la estructura de un nodo terminal, mostrada en la Figura 6.2. El nodo contiene hasta $n-1$ claves $K_1, \ldots, K_{n-1}$ y n punteros $P_1, \ldots, P_n$. Las claves se hallan ordenadas de modo que $K_i < K_{i+1}$. Para $1 \le i < n$, P_i apunta al bloque del archivo que contiene el registro cuya clave es K_i. El último puntero, P_n, encadena un nodo terminal con el siguiente. Esto permite recorrer el archivo secuencialmente en forma eficiente siguiendo la cadena de punteros P_n, aunque el archivo en sí no esté almacenado en ningún orden particular.

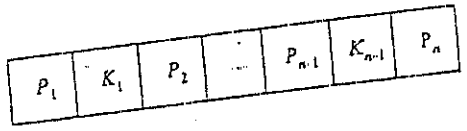


Figura 6.2
Estructura de un Nodo Terminal

Los nodos no-terminales tienen la misma estructura que los terminales, pero los punteros apuntan a los hijos de cada nodo en el árbol. Supongamos que un nodo no-terminal contiene d punteros, con $n/2 \le d \le n$. Para cada $1 < i < d$, el puntero P_i apunta al subárbol que contiene todas las claves mayores o iguales que K_{i-1} y menores que K_i. El puntero P_1 apunta al subárbol que contiene las claves menores que K_1, y P_d apunta al que contiene las claves mayores o iguales que K_{d-1}.

La Figura 6.3 muestra un ejemplo de un árbol-B de orden 3. Cada nodo contiene 1 ó 2 claves y 2 ó 3 punteros. Las líneas de puntos representan el encadenamiento de todos los nodos terminales. Notemos que la distancia de la raíz a cualquier nodo terminal es 2, mostrando que el árbol es balanceado.

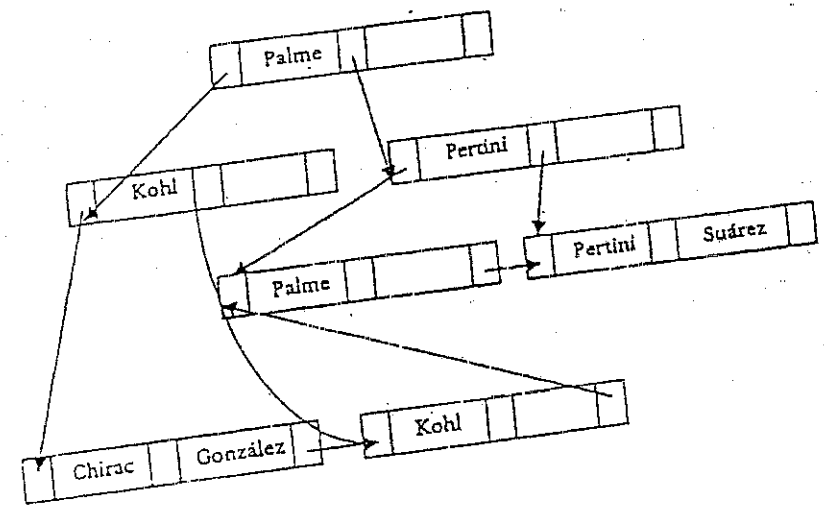


Figura 6.3
Un árbol-B de orden 3

El algoritmo de búsqueda es el siguiente. Supongamos que queremos encontrar el registro con clave K. Buscamos en la raíz la primera clave K_i tal que $K_i > K$. Seguimos el puntero P_i, bajando al próximo nivel del árbol. Si $K < K_1$, usamos el puntero P_i; si el nodo contiene d punteros, y $K > K_{d-1}$, usamos el puntero P_d. En el próximo nivel, repetimos el mismo proceso, y seguimos bajando en el árbol hasta llegar a un nodo terminal. Allí encontraremos un puntero que nos indica en qué bloque se halla el registro buscado.

Supongamos que cada nodo del árbol-B ocupa un bloque de disco. Esto nos dará valores típicos de 100 o más punteros por nodo. Vemos que la cantidad de accesos necesaria para encontrar un registro es uno más que la longitud del camino de la raíz a un nodo terminal. Gracias a la propiedad de que el árbol es balanceado, esta longitud es aproximadamente $\log_{n/2} M$, donde M es la cantidad de registros en el archivo. Por ejemplo, con un millón de registros y n = 200, tenemos alrededor de $1 + \log_{100} 10^6 = 4$ accesos por búsqueda.

La actualización es más complicada, ya que debe garantizar que se mantienen las dos propiedades que definen un árbol-B. Una inserción puede requerir la *partición* de un nodo en dos, y una baja puede causar que dos nodos se *combinen* en uno. Supongamos por ahora que esto no ocurre. En este caso, los algoritmos de alta y baja se definen así:

- **Alta:** Para insertar un registro con clave K, agregamos ese registro al final del archivo, o donde hubiera espacio, digamos en el bloque b. Ahora buscamos K en el árbol y ubicamos en qué nodo terminal aparecería. Insertamos la clave K y un puntero al bloque b en la posición correcta dentro del nodo terminal.
- **Baja:** Buscamos el registro a eliminar y damos de baja del nodo terminal el par de clave y puntero correspondiente.

Veamos ahora un caso en el cual es necesario partir un nodo. Supongamos que queremos insertar la clave "Craxi" en el árbol de la figura 6.3. La búsqueda nos indica que "Craxi" debería aparecer en el mismo nodo terminal que contiene "Chirac" y "González". Como $n = 3$, este nodo está lleno. Por lo tanto, lo particionamos creando un nuevo nodo. De las n claves, incluyendo la que estamos insertando, las primeras $n/2$ quedan en el nodo existente y las restantes se mudan al nuevo nodo. En el ejemplo, resulta un nodo con las claves "Chirac" y "Craxi" y otro con la clave "González". El nuevo nodo terminal, conteniendo una sola clave, debe ahora ser agregado al árbol. Para ello tomamos la mínima clave del nodo (en este ejemplo, la única clave en el nodo es "González") y la insertamos en el nodo padre del nodo que fue particionado. La Figura 6.4 muestra el resultado final de la inserción. Notemos que en este ejemplo fue posible insertar la clave "González" en el nodo que contenía la clave "Kohl" porque éste tenía espacio disponible. Si en cambio hubiera estado lleno, hubiéramos tenido que particionarlo tal como particionamos el nodo terminal; esto a su vez causa que una clave se deba agregar al padre del nodo particionado, lo cual puede causar una nueva partición al nivel superior, y así siguiendo recursivamente. En el peor caso, es posible que la cadena de particiones llegue hasta la raíz, en cuyo caso la raíz se particiona, se crea una nueva raíz y la profundidad del árbol se incrementa en 1.

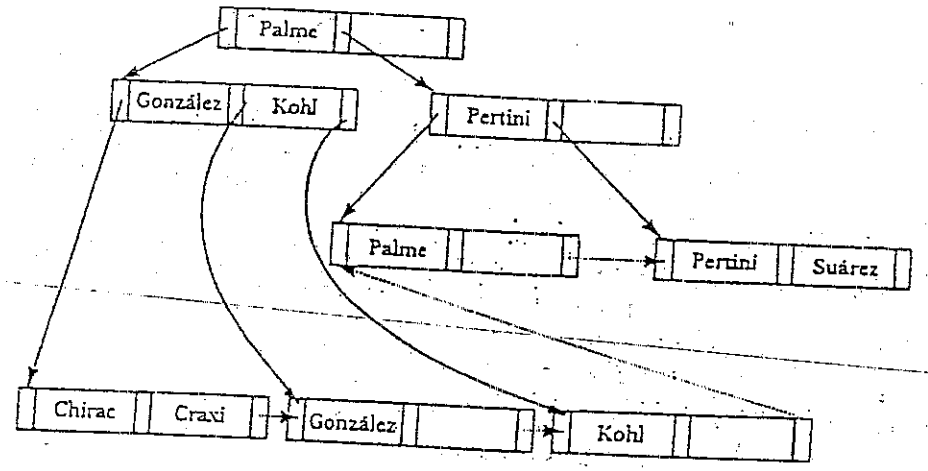


Figura 6.4
Resultado de Insertar *Craxi*

Supongamos ahora que queremos eliminar la clave "González" del árbol de la Figura 6.4. Al eliminar esta clave de su nodo terminal, éste quedaría vacío. Ya que $0 < n/2$, debemos eliminar este nodo del árbol. Para ello basta con eliminar el puntero que a él apunta desde su nodo padre. Esto deja dos punteros en el nodo padre, lo cual satisface la propiedad del árbol-B. El resultado es la Figura 6.5.

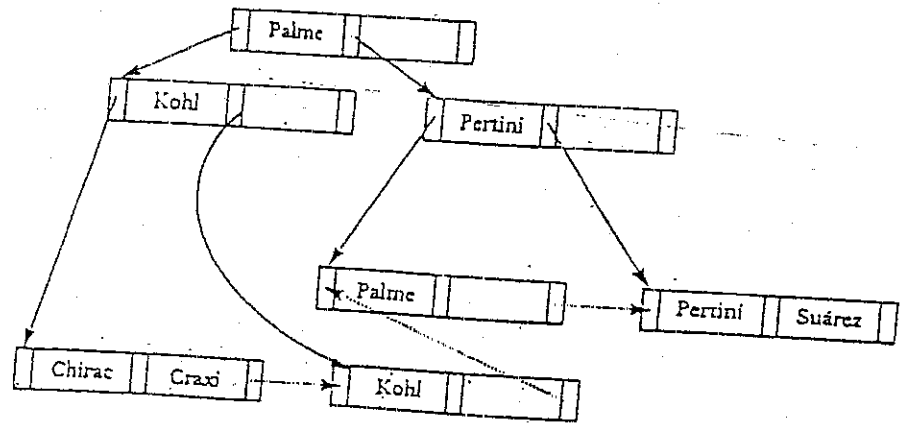


Figura 6.5
Resultado de eliminar *González*

Si en cambio la eliminación de una clave del nodo padre hubiera provocado que éste bajara del límite mínimo de cantidad de claves, hubiera sido necesario *combinar* este nodo con otro nodo al mismo nivel, ya sea convirtiendo dos nodos en uno o redistribuyendo las claves de manera que ambos nodos queden dentro de los límites permitidos. Por ejemplo, supongamos que queremos dar de baja la clave "Palme" del árbol de la Figura 6.5. La eliminación de "Palme" de su nodo terminal hace que éste quede vacío. Al eliminar el puntero al nodo vacío de su nodo padre, éste queda con un solo puntero, debajo del límite permitido. Consideremos el nodo "hermano" que contiene la única clave "Kohl". Podemos agregar a este nodo la clave "Pertini", combinando dos nodos en uno, y eliminar el nodo que quedó solamente con la clave "Pertini". Al eliminar este último la raíz se vuelve redundante, de modo que la eliminamos y la profundidad del árbol decrece en 1. La figura 6.6 muestra el resultado.

Aunque estas operaciones son de implementación relativamente complicada, no es difícil demostrar que su costo es logarítmico en el número de claves del archivo. De este modo obtenemos una estructura eficiente y flexible que se va reorganizando "de a poco" y por lo tanto no requiere periódicas reorganizaciones masivas.

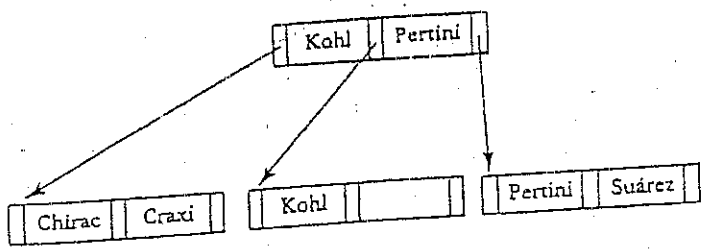


Figura 6.6
Resultado de eliminar *Palme*

Nota: existen múltiples variantes de estructuras de árboles-B. La que hemos descripto se llama generalmente *árbol-B+* en la literatura.

6.5. HASHING

En situaciones donde la cantidad de bloques a leer por cada acceso a registro se desea llevar al mínimo posible, es deseable evitar la búsqueda en estructuras adicionales como un índice o un árbol-B. El método más común de lograr esto es el uso de una *función de hashing* que permite, dada una clave, calcular inmediatamente el bloque de disco donde se halla el registro con esa clave.

En principio, esta función podría ser unívoca y muy simple: supongamos que existen N claves posibles, $K_0, \ldots, K_{N-1}$ y que en cada bloque se pueden almacenar R registros. Si almacenamos el registro con clave K_i en el bloque $B_{i/R}$ los registros con los R primeros valores de la clave irán al primer bloque, y así sucesivamente. El problema con este esquema es que el valor de N puede ser astronómico en comparación con la cantidad de registros M que esperamos almacenar. Por ejemplo, mientras existen solamente algunos miles de políticos europeos, el valor de N en este caso, suponiendo nombres de no más de 20 letras de largo, es de alrededor de 26^{20}, o sea varias decenas de millones. Por lo tanto, en general la idea es buscar una función f que envíe el conjunto de claves al conjunto de enteros $[0 \ldots M-1]$, donde M es el número de bloques que estimamos necesario para almacenar el archivo. Como $N >> M$, necesariamente esta función asigna el mismo valor a distintas claves y hará falta un mecanismo de resolución de *colisiones*, o sea, situaciones donde $f(K_i) = f(K_j)$.

Supongamos, por ejemplo, que queremos una función f que envíe el conjunto de nombres de políticos al conjunto $[0 \ldots 99]$. Podríamos probar la siguiente función: numeremos cada letra del 0 al 25 (a = 0, b = 1, etc.), suponiendo un alfabeto de 26 letras. Sea p_K el producto de los números asignados a la primera, quinta y última letra de una clave K; definimos $f(K) = p_K \bmod 100$. Éste es el resultado de aplicar esta función al conjunto de claves de la figura 6.3:

Palme	40
Kohl	24
Pertini	60
Chirac	00
González	00

Notamos que aun en esta pequeña muestra hemos obtenido una colisión con el valor 00. El problema es que cualquier clave que tenga la letra "a" en la primera, quinta, o última posición irá a parar al 00. El ejemplo sugiere que

el diseño de funciones de hashing es un ejercicio no trivial que no discutiremos aquí. Basta con notar que lo que se desea es una función que simule una asignación aleatoria con distribución uniforme de claves a bloques, de modo que ningún bloque se cargue excesivamente de registros.

Si, a pesar de esto, al insertar una clave *K* descubrimos que el bloque *f(K)* ya está lleno, recurrimos al método de resolución de colisiones; una solución común es crear un nuevo bloque y encadenarlo al existente usando punteros. Se forma así, para cada valor *i* en el rango de *f*, una lista de todos los bloques que contienen registros cuyas claves son enviadas a *i* por la función *f*. Cada lista de bloques se denomina un *balde*. El algoritmo de búsqueda para la clave *K* consiste por lo tanto en calcular el valor *f(K)* y luego recorrer la lista encadenada de bloques del balde que comienza en la dirección *f(K)*. El costo es entonces la longitud promedio de estas listas de bloques o tamaño medio de los baldes. Si la función tiene comportamiento uniforme, podemos controlar la cantidad de bloques asignada de modo de garantizar con muy alta probabilidad que la cantidad de bloques que se deben acceder para recuperar un registro no sea mayor que una pequeña constante independiente del tamaño del archivo. Si hacemos que esta cantidad, *B*, sea igual a la cantidad de bloques a almacenar en el archivo, el comportamiento aleatorio de la función hará que algunas listas crezcan más de lo deseado. Comúnmente se eligen valores de *B* bastante mayores que el tamaño del archivo, por ejemplo, alrededor del doble. Por supuesto, el costo de elegir un valor grande de *B* es el desperdicio de espacio, ya que debemos asignar *B* bloques aunque sólo vayamos a usar una fracción de B.

Una forma de implementar el método descripto es utilizar un directorio de baldes donde cada entrada contendrá la dirección del primer bloque de cada balde. La función de hash producirá cada vez la dirección de una entrada en el directorio.

6.6. Hashing Extensible

Los métodos como el de hashing descripto anteriormente, conocido como hashing *estático* tienen como objetivo, tal cual se expresó, poder leer un registro dado en la mínima cantidad posible de accesos al disco. Obviamente, la mínima cantidad de accesos es 1 y, como también se vio, con el método anterior es altamente probable que el objetivo no se cumpla. Más aún, cuando alguna lista de bloques encadenados, o balde, degenera y adquiere una longitud significativa, deberíamos reorganizar el archivo, con la finalidad de que la longitud promedio se mantenga baja. Esa reorganización implica aumentar la cantidad

de baldes y resulta sumamente costosa, ya que involucra a todos los registros del archivo. Este tipo de reorganización se denomina *global*.

A efectos de solucionar este problema, presentaremos un método que permite, independientemente del tamaño del archivo, recuperar un registro por medio de exactamente un acceso al disco y, eventualmente, un acceso adicional para leer un bloque del directorio. Por otra parte, las reorganizaciones que resultan del aumento en la cantidad de registros son *locales*, es decir, involucran sólo a los registros de uno o dos bloques. Este método, denominado Hashing Extensible es un ejemplo de hashing *dinámico* y es al hashing estático lo que el Árbol B⁺ es al método secuencial indexado.

El Hashing Extensible se caracteriza por utilizar dos estructuras separadas: una tabla hash, denominada *directorio* y un conjunto de páginas *hoja*. Las hojas almacenan las claves K junto con su información asociada I(K), la que puede ser un registro asociado con K o un puntero al registro que se encuentra almacenado en otro archivo.

Si K es una clave y h una función de hash, h(K) = K' se denomina la pseudoclave asociada con K y se caracteriza por una secuencia de bits de longitud fija.

El directorio es un arreglo que tiene 2^d entradas, donde d se denomina la *profundidad del directorio* y es conservada en su encabezamiento. Si M es la cantidad máxima de bits de la pseudoclave, luego $1 \leq d \leq M$. Cada entrada del directorio contiene un puntero a una página hoja. A cada entrada se le asigna una dirección dada por la configuración de d bits. Así, la primer entrada se identifica por d bits 0, la segunda por d – 1 bits 0 y el último bit 1, hasta la última entrada definida por d bits 1. La función de hash h nos asegura que el i-ésimo bit de K' será 0 con probabilidad 0.5, independientemente de los valores que adopten los restantes bits, como así también de la distribución de los datos.

Luego, una entrada caracterizada por una determinada configuración de bits tendrá un puntero a una página hoja conteniendo todas las claves K para las que h(K) = K', tales que K' comienza con exactamente la configuración de bits que caracteriza a esa entrada.

Por su parte, cada página hoja tiene una *profundidad local* d', donde $d' \leq d$. La profundidad local expresa que para todas las claves contenidas en una página dada sus pseudoclaves K' coinciden en los d' primeros bits.

Veamos ahora un ejemplo para clarificar lo expuesto: supongamos d = 3, luego tendremos 8 entradas, cada una caracterizada por 000, 001, 010, 011, 100, 101, 110 y 111. La Figura 6.7 describe una situación que tomamos como inicial.

Si se desea localizar un registro con clave V calculamos h(V) = V'. Se considera a V' como una secuencia de bits. De V' se toman los d primeros

Procedimiento:

i. calcular K' = h(K);
ii. leer la profundidad del directorio d;
iii. tomar los d primeros bits de K', interpretarlos como un entero base 2 y acceder a ese número de entrada en el directorio. Cada entrada contiene el puntero a una página hoja y la cantidad de registros que esa página contiene;
iv. leer el puntero y por su intermedio recuperar la página hoja P a memoria;
v. buscar dentro de la página P, en memoria, el registro con clave K;
vi. si el registro no está en P emitir mensaje de "No Hallado";
vii. end.

Algoritmo de Inserción

Entrada: Registro con clave K e información asociada I(K).
Salida: Aviso de inserción cumplida o no.

Procedimiento:

i. aplicar Algoritmo de Búsqueda con clave K;
ii. if existe registro con clave K then return ("No se puede insertar registro con clave duplicada K");
iii. if existe espacio suficiente para insertar el registro en página P
then begin
iv. insertar el registro en P en el lugar en el que el Algoritmo de Búsqueda trató de hallarlo;
v. actualizar la cantidad de registros en el directorio;
vi. return ("inserción exitosa");
vii. end;
viii. Obtener una nueva página P' {no hay espacio suficiente en la página original P};
ix. almacenar todos los registros contenidos originalmente en P más el nuevo K, I(K) en un área temporaria Q;

x. incrementar la profundidad local d' de P en 1, asignar esa profundidad a P y P';
xi. borrar las páginas P y P', salvo las profundidades;
xii. if d' > d {la profundidad local de alguna página supera la profundidad global}
then begin
xiii. incrementar la profundidad global d en 1;
xiv. duplicar el tamaño del directorio;
xv. actualizar los punteros a las páginas hoja;
xvi. end;
xvii. *INSERTAR* todos los registros contenidos en Q; {llamado recursivo al Algoritmo de Inserción}
xviii. end.

Algoritmo de Eliminación

Entrada: Clave K del registro a eliminar.
Salida: Aviso de eliminación cumplida o no

Procedimiento:

i. aplicar Algoritmo de Búsqueda con clave K;
ii. if no existe registro con clave K
then
iii. return ("no existe el registro que se desea eliminar");
iv. eliminar el registro marcándolo como *eliminado*;
v. if la cantidad de registros de P más la cantidad de registros de su página "hermana" es menor que un umbral prefijado
then begin {se une el contenido de las dos páginas en una sola};
vi. copiar el contenido de las dos páginas a un almacenamiento temporario Q;
vii. liberar una de las dos páginas;
viii. modificar los punteros para que apunten a la página restante P;

```
ix                  reducir la profundidad de la página P en 1;
x                   borrar la página P, salvo su profundidad;
xi                  INSERTAR en P los registros contenidos en Q;
xii             end;
xiii    if todos los punteros en el directorio están al menos duplicados
            then begin
xiv                 reducir la profundidad global en 1;
xv                  dividir por 2 el tamaño del directorio;
xvi                 actualizar los punteros;
xvii        end;
xviii   end
```

6.7. Comparación de los Métodos

Casi todos los sistemas de bases de datos usan alguna variante de indexación incluyendo árboles-B o hashing, o a veces ambos métodos. En general, la decisión de qué estructura física implementar depende de las características de uso del sistema: la frecuencia relativa de actualizaciones respecto de consultas, la necesidad o no de un tiempo máximo de respuesta garantizado, el tipo de consultas, etc.

Consideremos en particular el tipo de consultas. Para consultas de la forma

```
select K, A1, ..., An
from R
where K = c
```

o sea, recuperar un registro dado el valor de su clave, el método de hashing es claramente superior en tiempo de respuesta. La única ventaja del indexado en este tipo de consulta es que garantiza un costo logarítmico aun en el peor caso, mientras que el hashing (estático) sólo garantiza el costo constante en un sentido probabilístico, y el peor caso puede llegar a ser lineal en el tamaño del archivo (por ejemplo, cuando todas las claves son enviadas al mismo valor por la función). En la práctica, la probabilidad de que esto ocurra puede hacerse suficientemente baja que esto no es un problema. El hashing extensible, por otra parte, garantiza un costo constante de un acceso.

Supongamos en cambio que las consultas más frecuentes requieren un barrido secuencial del archivo, por ejemplo:

```
select *
from R
order by K
```

Los métodos de indexación permiten recorrer el archivo en orden de clave sin ningún costo extra; si en cambio usamos hashing, la única forma es clasificar el archivo. También en las consultas llamadas *de rango*, por ejemplo:

```
select *
from R
where K ≤ c1 and K ≥ c2
```

el indexado es superior; basta con encontrar el bloque donde se halla la clave c_2 y recorrer el archivo secuencialmente desde allí hasta superar el valor c_1. Si usamos hashing, no hay otra forma de evaluar esta consulta que recorrer toda la relación R verificando qué tuplas están en el rango deseado.

En sistemas que soportan tanto índices o árboles-B como hashing, como por ejemplo *Ingres*, el lenguaje de consulta da al usuario la posibilidad de especificar, al crear una relación, qué tipo de estructura usar para accederla.

6.8. Accesos por Campos No-Clave

Hasta ahora, hemos considerado solamente accesos donde se desea recuperar el valor de un registro dado el valor de su clave primaria. Existen por supuesto muchas consultas que no son de esta forma, sino que requieren encontrar todos los registros tales que uno o más de sus atributos satisfacen una condición. El caso más simple es una condición de igualdad sobre un campo no-clave, por ejemplo:

```
select *
from Médico
where espec = "card"
```

Las técnicas de indexación y hashing que hemos visto se aplican directamente a este tipo de consultas. Es decir, así como podemos crear un índice para el campo clave, podemos también crear uno para cualquier otro campo. La diferencia es que, si el atributo no es clave, existen potencialmente múltiples registros con un mismo valor y las estructuras deben modificarse para permitir esto, lo cual no es difícil.

Existen técnicas más sofisticadas que permiten acceso rápido a registros dado el valor de dos o más atributos: por ejemplo, árboles multidimensionales, reticulado *(grid)*, hashing particionado. No haremos aquí más que mencionarlas.

EJERCICIOS

E1. Se dispone de un espacio de almacenamiento en disco donde cada bloque es de 512 bytes. En dicho espacio se almacenan tablas e índices. Los bloques destinados a tablas cuentan con un encabezado de 15 bytes más una lista de punteros a las tuplas, con 2 bytes por puntero. Los bloques destinados a índices tienen un encabezado de 10 bytes y su estructura interna corresponde a nodos de un árbol-B+. Los números de bloque insumen 4 bytes.
El almacenamiento se ha ocupado con:

- Una tabla S de 695 tuplas de 64 bytes cada una, con la siguiente composición: B(4 bytes) clave primaria, C(30 bytes), D(30 bytes).
- Un índice sobre la columna B de S.
- Una tabla R de 165.510 tuplas de 8 bytes cada una. La tabla tiene el siguiente esquema: A(4 bytes) clave primaria, B(4 bytes) clave foránea referenciando la tabla S.
- Un índice sobre la columna A de R.

Se pide determinar:

i. ¿Cuántos accesos demanda la inserción de una tupla en la tabla R en el peor caso?

ii. ¿Cuántos accesos demandaría realizar la consulta


```
select C, D
  from R, S
    where R.B = S.B and R.A = 25;
```

si se toman los índices como camino de acceso a las tablas?

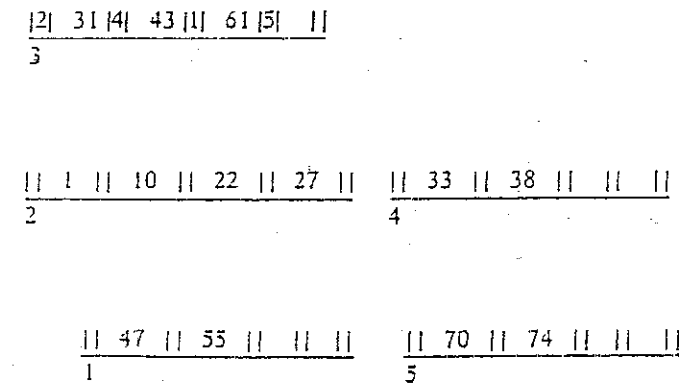
E2. Sea un árbol-B de 3 niveles. Las claves son números enteros y la capacidad máxima de un nodo son 4 claves. El árbol se encuentra en el siguiente estado:

- La raíz tiene una única clave con valor 110.
- Los dos niveles siguientes tienen las claves 4, 25, 33, 37, 45, 55, 58, 66, 74, 90, 93, 106, 115, 120, 133, 137, 141, 155, 162, 189, 206.

i. Muestre, si existe, una estructura posible para el árbol que, cumpliendo con las dos condiciones citadas, tenga al menos una de sus hojas llenas.

ii. Muestre, si existe, una estructura posible para el árbol que, cumpliendo con las dos condiciones citadas, pierda un nivel al eliminar alguna de sus claves, señalándola (considerar que el algoritmo intenta reorganizar si es posible; en caso contrario busca fusionar con el hermano izquierdo y sólo si esto no es posible fusiona con el hermano derecho).

E3. Se han ingresado claves enteras en el árbol B (de orden 5) siguiente:



i. Indique una secuencia de enteros cuya inserción a partir del árbol-B nulo produzca este árbol

ii. Muestre el árbol binario de búsqueda que corresponde a este árbol-B, si se supone una búsqueda secuencial en los bloques.

iii. Determinar la cantidad de accesos a bloques que demanda insertar la clave 29 en el árbol (incluyendo tanto operaciones de escritura como de lectura).

iv. Mostrar cómo quedaría el árbol luego de borrar el 74, si siempre se prefiere reorganizar primero localmente y, si esto no fuese posible, fusión con el hermano izquierdo a continuación y en última instancia, fusión con el hermano derecho

v. Muestre una secuencia de enteros que, de ser agregados al árbol original produzcan aumento en la altura

vi. Muestre una secuencia de enteros presentes en el árbol original que, de ser eliminados, produzcan una disminución en la altura.

E4. ¿Existe un árbol-B equivalente al del ejercicio anterior (visto como conjunto de enteros) pero con mayor altura?

E5. Produzca un árbol-B* con conjunto secuencia para almacenar tuplas con conjunto de claves igual al conjunto del ejercicio anterior, partiendo del árbol nulo. Los bloques del conjunto secuencia admiten 3 claves solamente.

E6. Defina un método que permita, dada una secuencia de tuplas ordenada por orden ascendente de la clave, insertarlas en un árbol-B* nulo, pero garantizando que los bloques terminales estén todos llenos con excepción del último. Asuma *d* punteros en los bloques no terminales y *e* punteros a tuplas en los nodos terminales

E7. Sea una organización basada en hashing con N baldes. Suponga que la función de hash para una clave v es $h(v) = g(v) \bmod N$, donde $g(v)$ es una función que asume valores enteros mucho mayores que N. Suponga también que el sistema es capaz de detectar la existencia de baldes con muchos bloques y decidir si es necesario una reorganización completa del archivo.
Si en una reorganización el número de baldes se triplica, se pide:

i. determinar en qué baldes del archivo reorganizado se encontrarán los registros que estaban originalmente en balde i;

ii. ¿qué sucedería si $g(v)$ fuera menor que N para todo v?

E8. Sea un archivo organizado según el método de *hashing extensible*, con las siguientes características: la función de hash es dividir por 41 y tomar el resto, y la cantidad máxima de registros por bloque se determina en 3.

i. ¿Cuántos bits asignaría a la pseudoclave?
Suponga que las claves de los registros a ingresar son numéricas y se encuentran entre 500 y 1000. Elija las claves a su criterio de manera que:

ii. los 3 primeros registros insertados vayan al mismo bloque;

iii. a partir de allí que el directorio se incremente en una unidad cada 2 registros insertados;

iv. con este método de inserción, ¿cuántos registros se habrán insertado cuando la profundidad del directorio sea 5? ¿Cuántos cuando sea 6?;

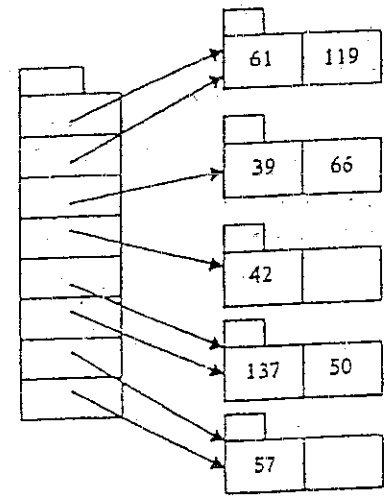
v. muestre el estado del archivo cuando la profundidad sea 5;

vi. ¿hay alguna secuencia de eliminación de registros tal que haga que al eliminarse un registro la profundidad del directorio se reduzca en más de una unidad?

E9. Sea un sistema de hashing extensible con las siguientes características: cada bloque puede tener hasta 2 registros; la función de hash es dividir por 29 y tomar el resto.

i. ¿Cuántos bits son necesarios para la pseudoclave?

ii. En un cierto instante el estado del sistema es el siguiente:



Complete los valores de profundidad del directorio y las profundidades locales.

iii. A continuación se enumeran una serie de operaciones sobre el archivo, efectuadas, cada una de ellas, con el archivo en el estado representado en el punto ii, y una serie de posibles consecuencias. Indique qué consecuencias implica cada una de estas operaciones.

Operaciones: insertar el registro con clave:

a) 293 b) 166 c) 35 d) 110 e) 49

Consecuencias:

1. Duplica la cantidad de entradas del directorio.
2. Duplica la profundidad del directorio.
3. Reduce la profundidad del directorio.
4. Produce modificaciones en el directorio, sin cambiar de tamaño.
5. No produce ningún cambio en el directorio.
6. No puede ser satisfecha. (Sugiera una forma de modificarlo a fin de que pueda ser satisfecha.)
7. Lleva la profundidad del directorio a 5.
8. Entre las consecuencias del punto anterior hay una que es "imposible" (no existe ninguna operación que al aplicarse sobre este archivo produzca esa consecuencia). ¿Cuál es? Dar los motivos.

NOTAS BIBLIOGRÁFICAS

El texto de T. J. Teorey [198] trata a fondo los problemas de selección de estructuras físicas para implementar bases de datos. También contiene información útil sobre este tema, y sobre bases de datos en general, el libro de G. Wiederhold [213] (hay traducción al español). Sobre estructuras de datos en general existen una amplia bibliografía, de la cual destacamos la magnífica obra de D. E. Knuth [141].

El artículo original sobre el árbol-B es el de Bayer-McCreight [18]. Otros artículos interesantes son el de Comer [69] y el de Held-Stonebraker [127].

En el caso de hashing extensible, el artículo original es el de Fagin y otros [87].

CAPÍTULO 7

PROCESAMIENTO DE CONSULTAS

7.1. INTRODUCCIÓN

Dada una consulta en un lenguaje como SQL, el sistema de base de datos debe de alguna manera generar las tuplas de la respuesta. De la eficacia de los métodos usados en esta tarea dependerá en gran medida la performance total del sistema. Un lenguaje de consulta relacional da al usuario el poder de especificar en forma sucinta consultas cuya implementación puede ser muy costosa a menos que el sistema elija cuidadosamente el método más eficiente para procesarlas. La diferencia en costo entre un buen y un mal método puede ser de varios órdenes de magnitud.

EJEMPLO 7.1: Consideremos el esquema relacional del Capítulo 5, donde teníamos:

Alumno (Anumero, Anombre)
Examen (Anumero, Materia, Efecha, Nota)

Supongamos que debemos procesar la consulta "hallar los nombres de los alumnos que sacaron más de 8 en algún examen", o sea, en SQL:

```
select Anombre
from Alumno, Examen
where Alumno.Anumero = Examen.Anumero
  and Nota > 8;
```

Traduciendo esta consulta al álgebra relacional, obtenemos

$$\pi_{Anombre}(\sigma_{Nota>8}(\sigma_{I=3}(Alumno \times Examen)))$$

Supongamos que existen 5000 tuplas de Alumno, almacenadas a 20 por bloque físico, y 100 000 tuplas de Examen, almacenadas a 10 por bloque. El método ingenuo de evaluar esta expresión algebraica consiste en formar el producto cartesiano *Alumno* × *Examen* y luego aplicarle las selecciones y la proyección. El cómputo del producto requiere leer 250 × 10.000 = 2.500.000 bloques y escribir otros tantos, generando un resultado intermedio de 2 500.000 bloques que luego debe ser barrido para seleccionar las tuplas relevantes; el costo total en cantidad de accesos a bloque resulta de alrededor de 7.500.000.

Una estrategia más sofisticada observará que el producto cartesiano seguido de la selección es en realidad una operación de junta que puede realizarse en forma mucho más eficiente. Supongamos que una relación de n bloques puede ser puesta en orden de acuerdo con el valor de un atributo con costo $n\log_2 n$. Ordenamos las relaciones Alumno y Examen de acuerdo con el valor de Anúmero; el costo del ordenamiento es de 250log250 + 10.000log10000 aproximadamente igual a 140.000. Ahora podemos implementar la junta barriendo secuencialmente ambas relaciones en forma similar a una operación de "merge" de archivos; cuando encontramos un par de tuplas que concuerdan en número de alumno y satisfacen la condición de que la nota sea mayor que 8, emitimos el nombre del alumno. Suponiendo que la cuarta parte de los alumnos van a satisfacer la condición, el costo de este paso es de 10.250 accesos de lectura y 50 accesos de escritura. El costo total resulta así de alrededor de 150 000 accesos, 50 veces menor que el costo del método ingenuo. Notemos que el costo se podría haber reducido aun más si hubiéramos combinado el ordenamiento de la relación *Examen* con la selección de las tuplas con nota mayor que 8; si suponemos que un décimo de las tuplas satisfacen esta condición, el resultado de este paso es una relación *Examen* con sólo 1000 bloques en lugar de 10.000, y el costo total se reduce a 141.000 accesos.

Costos de Procesamiento

El objetivo del procesamiento de consultas es la minimización del costo de la producción de una respuesta. El costo a minimizar se compone de distintos factores, entre ellos los siguientes:

- **Comunicaciones:** El costo de transmitir datos desde donde estén almacenados hasta donde se realice el cómputo y presentación de los resultados. Por ejemplo, en una base de datos geográficamente distribuida, el procesador de consultas tratará de usar en lo posible datos almacenados localmente en el lugar donde se hace la consulta y minimizar la cantidad de datos que deben ser transferidos mediante las líneas de comunicación. Como en este libro no trataremos los problemas de bases de datos distribuidas, ignoraremos de ahora en más los costos de comunicaciones.
- **Acceso a memoria secundaria:** El costo de transferir bloques de datos de memoria secundaria a memoria principal y viceversa. Los factores que influyen sobre este costo son: cantidad de tuplas a ser transferidas, agrupamiento de tuplas en bloques físicos, cantidad de espacio de buffers disponible en memoria principal, y velocidad de acceso de los dispositivos de memoria secundaria.
- **Tiempo de Procesador:** El costo de utilizar la unidad central de procesamiento.

Como ya hemos indicado, la tecnología actual implica que en la mayoría de los casos –pero no en todos– el tiempo de acceso a memoria secundaria dominará el tiempo de uso del procesador; por lo tanto, los sistemas de base de datos consideran ventajoso gastar tiempo de procesador en la búsqueda de estrategias que disminuyan la cantidad de bloques a acceder.

Etapas en el Procesamiento de Consultas

El primer paso en la ejecución de una consulta es su traducción a una forma interna que sea fácilmente manipulable. Por ejemplo, es común transformar una consulta en una expresión en álgebra o cálculo relacionales, representadas mediante un árbol de sintaxis. Las técnicas para realizar esta transformación son similares a las que utiliza el analizador sintáctico de un compilador. A la vez, el analizador verifica la corrección de la sintaxis de la consulta, valida los nombres de relaciones y atributos utilizados, etc.

Una vez realizado el análisis sintáctico, comienza el proceso llamado *optimización de consultas*. Éste se compone de dos fases: en la primera fase, se intenta transformar la forma interna de la consulta en otra equivalente pero menos costosa de ejecutar. Por ejemplo, dada una expresión en álgebra rela-

cional, se trata de transformarla en otra expresión equivalente que sea más fácilmente computable. Esta primera fase es independiente de las estructuras físicas de almacenamiento utilizadas. La segunda fase toma el resultado de la primera y genera, tomando en cuenta las estructuras físicas tales como índices y tablas de hashing, una estrategia detallada para la ejecución de la consulta; esta estrategia suele llamarse un *plan de acceso*.

7.2. TRANSFORMACIÓN DE EXPRESIONES ALGEBRAICAS

Como sabemos, una misma consulta puede expresarse en muchas formas distintas en álgebra relacional; algunas de éstas pueden ser substancialmente más fáciles de ejecutar que otras. El objetivo de la primera fase del proceso de optimización de consultas es la selección de una expresión algebraica equivalente a la dada y eficientemente computable. En esta sección presentaremos reglas que pueden usarse en este proceso de transformación.

Selección

Como regla general, trataremos de aplicar los operadores de selección tan pronto como sea posible; esto reducirá el tamaño de los resultados intermedios.

EJEMPLO 7.2: Con el mismo esquema del ejemplo anterior, consideremos la consulta: "¿qué alumnos obtuvieron más de 8 en un examen tomado por el Prof. Sandrini?". La expresión algebraica

$$\pi_I(\sigma_{Nota>8}(\sigma_{Pnom=Sandrini}(Alumno \times Examen \times Prof)))$$

requiere la construcción de un resultado intermedio de gran tamaño, la junta

$$Alumno \bowtie Examen \bowtie Prof.$$

De este resultado, sólo una fracción de las tuplas será utilizada para obtener la respuesta final. Si en cambio seleccionamos primero las tuplas de Examen con Nota>8 y las de Prof con Pnom=Sandrini, obtenemos un ahorro considerable. La expresión algebraica que corresponde, equivalente a la original, es:

$$\pi_I(Alumno \bowtie \sigma_{Nota>8}(Examen) \bowtie \sigma_{Pnom=Sandrini}(Prof)).$$

Junta Natural

Otra forma de reducir el tamaño de los resultados intermedios consiste en usar un ordenamiento óptimo de las operaciones de junta. Como la junta natural es asociativa, sabemos que, para tres relaciones cualesquiera *R*, *S*, y *T*, se cumple que

$$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T.$$

El costo de evaluar la expresión de la izquierda puede, sin embargo, ser muy diferente del de evaluar la de la derecha. Por ejemplo, consideremos la junta

$$Alumno \bowtie Profesor \bowtie \sigma_{Nota>8}(Examen).$$

Si realizamos primero la junta *Alumno* |x| *Profesor*, vamos a obtener simplemente el producto cartesiano de todos los alumnos con todos los profesores, ya que estas dos relaciones no tienen ningún atributo en común. Sólo una fracción de las tuplas en este resultado intermedio sobrevivirán la operación de junta con $\sigma_{Nota>8}(Examen)$. En cambio, si realizamos primero la junta $Profesor \bowtie \sigma_{Nota>8}(Examen)$, el resultado tendrá la misma cardinalidad que el segundo operando (suponiendo que un solo profesor toma cada examen), y ésta es también la cardinalidad del resultado final (salvo por los alumnos que aparezcan duplicados), de modo que no hemos generado tuplas "inútiles". Notemos que como la junta es también conmutativa, podríamos igualmente haber tomado primero la junta $Alumno \bowtie \sigma_{Nota>8}(Examen)$, con el mismo resultado.

Proyección

Así como la selección reduce la cantidad de tuplas que aparecen en los resultados intermedios, la proyección reduce el tamaño de cada tupla y por lo tanto el tamaño total de los resultados intermedios. Por lo tanto, es ventajoso también, en general, tratar de aplicar las proyecciones lo más temprano posible. Por ejemplo, en la expresión

$$\pi_I(Alumno \bowtie (Profesor \bowtie \sigma_{Nota>8}(Examen)))$$

sólo vamos a usar el atributo *Anúmero* de la junta *Profesor* |×| $\sigma_{Nota>8}(Examen)$; por lo tanto sólo necesitamos en la junta el atributo *Anúmero* más todo atributo común a *Profesor* y a *Examen*, con lo cual podemos usar

$$\pi_{Materia\ Efecha}(Profesor)$$

en lugar de *Profesor*, y

$$\pi_{Anúmero,\ Materia,\ Efecha}(\sigma_{Nota>8}(Examen))$$

en lugar de $\sigma_{Nota>8}(Examen)$. Si lo deseamos, también podemos eliminar de la junta de *Profesor* y *Examen* todos los atributos salvo *Anúmero* antes de realizar la junta con *Alumno*.

Operaciones Booleanas

Tal como ocurre con la junta natural las operaciones de unión y diferencia gozan de propiedades que permiten efectuar las transformaciones sugeridas por las siguientes identidades:

$$\sigma_F(R \cup S) = \sigma_F(R) \cup \sigma_F(S)$$

$$\sigma_F(R - S) = \sigma_F(R) - S = \sigma_F(R) - \sigma_F(S)$$

$$(R \cup S) \cup T = R \cup (S \cup T)$$

$$R \cup S = S \cup R$$

7.3. Estimaciones del Costo de Procesamiento

Para seleccionar una estrategia de ejecución de una consulta entre las muchas posibles el optimizador de consultas debe poder estimar el costo de una estrategia, medido en cantidad de accesos a memoria secundaria. Este costo depende por supuesto de las características de los datos; en primer lugar, es necesario conocer:

cant de tupla en la relac

1. la cardinalidad de cada relación R, n_R;
2. la longitud (en bytes) de cada tupla de R, l_R;
3. para cada atributo A y cada relación R, la cardinalidad de $\pi_A(R)$, o sea, la cantidad de valores distintos de A que aparecen en R; llamemos a esta cantidad $V(A, R)$.

Usando esta información, podemos estimar el tamaño de cualquier resultado producido por una expresión algebraica. Comencemos con el producto cartesiano: la cardinalidad de $R \times S$ es fácilmente estimable como $n_R \times n_S$, y cada tupla del producto cartesiano ocupa $l_r + l_s$ bytes.

Supongamos que queremos estimar, dado n_R, la cardinalidad de

$$\sigma_{A=k}(R),$$

donde k es una constante. Si suponemos que los valores de A aparecen con distribución uniforme, o sea, cada valor de A aparece aproximadamente con la misma frecuencia, la cardinalidad de $\sigma_{A=k}(R)$ es aproximadamente igual a

$$n_R / V(A, R).$$

En general, esta suposición no es realista; por ejemplo, si R es *Examen* y A es *Nota*, sabemos que los valores de A no aparecerán uniformemente distribuidos, ya que algunas notas son mucho más comunes que otras. Dada la dificultad de obtener información confiable sobre la distribución de valores de un atributo, en la práctica los optimizadores de consultas tienden a usar la suposición de distribución uniforme.

La estimación del tamaño de una junta es algo más complicada que la del producto cartesiano. Sean R y S dos relaciones que tienen en común el conjunto de atributos X. Si $X = \emptyset$, la junta se reduce a un producto cartesiano; supongamos por lo tanto que X no es el conjunto vacío. Si X es una clave para una de las relaciones, digamos R, cada tupla de S concuerda con a lo sumo una tupla de R; por lo tanto, la cardinalidad de R |×| S está acotada superiormente por n_S. Si en cambio X no es una clave para ninguna de las dos relaciones, el problema es más complicado.

Supongamos por simplicidad, que la intersección de R y S contiene un solo atributo, que llamaremos A, o sea, $X = \{A\}$. Supongamos, además, que los valores de A se hallan uniformemente distribuidos. Para cada tupla de R existen

$$n_S / V(A, S)$$

tuplas en S con el mismo valor de A. Por lo tanto, la cardinalidad de $R \bowtie S$ se puede estimar como

$$n_R\, n_S / V(A, S).$$

Notemos que podríamos igualmente usar la expresión

$$n_R\, n_S / V(A, R).$$

Si estas dos expresiones difieren significativamente debido a una diferencia entre $V(A, R)$ y $V(A, S)$, es plausible suponer que una de las relaciones contiene sólo algunos de los valores de A que aparecen en la otra, y en este caso la menor de las dos estimaciones será más acertada. Si en cambio R y S tienen pocos valores de A en común, ambas estimaciones serán demasiado altas; necesitamos más información para poder hacer una estimación ajustada.

Uso de índices

Como explicamos en el capítulo anterior, el uso de estructuras de datos adicionales, tales como índices y tablas de hashing, es esencial para obtener tiempos de respuesta adecuados en el procesamiento de consultas. Hasta el momento no hemos considerado el efecto de estas estructuras en la estimación de costos de procesamiento, pero es obvio que su presencia influye fuertemente en la selección de una estrategia de resolución de una consulta.

Para simplificar el problema, consideremos simplemente una consulta equivalente a una selección algebraica seguida de proyección, o sea, en SQL:

```
select A1, ..., An
from R
where P1 and P2 and ... and Pn
```

donde los P_i's son condiciones sobre los atributos de R. El método que describiremos para hallar una estrategia de resolución de estas consultas está basado en el que utiliza el Sistema R. Hay dos factores que considerar:

- Presencia de índices: Si uno de los P_i's es de la forma $A = k$, y existe un índice sobre el atributo A, es probablemente ventajoso obtener primero todas las tuplas de R que satisfacen esta condición, usando el índice, y verificar cuáles de ellas satisfacen los demás P_i's. El costo de este plan de acceso, medido en cantidad de bloques, será el costo de acceder el índice más el costo de acceder las tuplas; este último varía ampliamente según como estén físicamente repartidas las tuplas en el disco, como veremos inmediatamente.
- Tipo de índices: Supongamos que el sistema permite almacenar relaciones usando agrupamiento, como lo definimos en la Sección 6.3, de modo que las tuplas de algunas relaciones pueden aparecer densamente cargadas en bloques y otras completamente esparcidas. En el ejemplo que dimos en la Sección 6.3, las tuplas de la relación Médico aparecen almacenadas en forma compacta, mientras que las de la relación Hospital se hallan esparcidas. Definamos un *índice de agrupamiento (clustering index)* como un índice sobre un atributo tal que las tuplas que tienen el mismo valor en ese atributo tienden a aparecer físicamente contiguas en el disco. Por ejemplo, un índice sobre el atributo *hosp* en la relación Médico de la Sección 6.4 es un índice de agrupamiento, ya que las tuplas de Médico con el mismo valor de *hosp* tenderán a aparecer en forma contigua. Otro ejemplo sería cualquier relación que esté almacenada físicamente en orden de cierto atributo A: un índice sobre A será entonces un índice de agrupamiento para esa relación. La presencia de un índice de agrupamiento nos permite obtener todas las tuplas con un valor dado en el atributo indexado, usando una cantidad total de accesos cercana al mínimo número de bloques necesario para almacenar esas tuplas. En cambio, un índice que no sea de agrupamiento puede requerir un número de accesos igual al número de tuplas a obtener.

En lugar de dar una descripción detallada del método de generación de un plan de acceso, veamos un ejemplo. Sea la consulta:

```
select nombre
from Medico
  where espec = psiq' and hosp = 45
    and número < 400;
```

O sea, los nombres de los psiquiatras que trabajan en el hospital número 45 y cuyo código es menor que 400. Supongamos que tenemos las siguientes estadísticas sobre la relación Médico:

- En cada bloque caben 20 tuplas.
- *V(hosp, Médico)* = 25.
- *V(espec, Médico)* = 20.
- *V(número, Médico)* = 5000.
- $n_{Médico}$ = 5000.

no es dato

Supongamos que existen dos índices en forma de árboles-B: un índice de agrupamiento sobre el atributo *hosp* y uno no de agrupamiento sobre el atributo *espec*. Supongamos también que todos los atributos aparecen uniformemente distribuidos.

Tenemos que elegir entre los dos índices disponibles para obtener el conjunto inicial de tuplas, un subconjunto del cual formará la respuesta. Supongamos que comenzamos usando el índice sobre *hosp* para extraer el conjunto de tuplas con *hosp* = 45. Como éste es un índice de agrupamiento, la cantidad de bloques a acceder será igual al cociente entre la cantidad de tuplas con *hosp* = 45 y la cantidad de tuplas por bloque. A este número deberemos sumar el costo de acceder el índice para obtener el costo total de procesamiento. Dada la suposición de uniformidad, la cantidad de tuplas con el número de hospital buscado es 5000/25 = 200. Como la cantidad de tuplas por bloque es 20, el costo en número de accesos resulta igual a 200/20 = 10. Notemos que a medida que obtenemos estas tuplas podemos ir verificando si satisfacen las otras dos condiciones, por lo que el costo total de la consulta usando este método es de 10 accesos más el costo de acceder al índice. Supongamos que el árbol-B contiene entre 5 y 10 punteros por nodo, y que cada puntero de un nodo terminal apunta al primer bloque del archivo que contiene registros con el valor asociado de *hosp*. La cantidad de punteros necesaria entonces es igual a la cantidad de valores distintos del atributo, o sea 25, lo que implica un mínimo de 3 nodos terminales y un máximo de 5, con lo que la profundidad del árbol será igual a 1, o sea que el costo de acceder el índice es de 2 accesos. El costo total de esta estrategia resulta entonces de 12 accesos.

Si en cambio usamos el índice sobre *espec*, el cálculo del costo es el siguiente. La cantidad de tuplas con especialidad psiquiatría será aproximadamente 5000/50 = 100. Como el índice no es de agrupamiento la cantidad

de bloques a acceder será también del orden de 100. Sin necesidad de calcular los accesos al índice, que serán 2 ó 3, vemos que es más conveniente por un orden de magnitud usar el índice sobre *hosp* en lugar del de *espec*. Notemos que, si ninguno de los índice hubiera sido de agrupamiento, hubiéramos llegado a la conclusión contraria, ya que el costo usando *espec* sería del orden de 100 contra 200 para *hosp*.

Finalmente, notemos que no es conveniente acceder primero la tuplas que satisfacen la condición *número < 400*, ya que esto requeriría una lectura secuencial de todas las tuplas de la relación, con un costo de 5000/20 = 250 accesos.

Generalizando a partir de este ejemplo, el método seguido en el Sistema R y otros sistemas similares en la optimización de consulta de este tipo se puede resumir como sigue. Consideremos todas la estrategias que consisten en una de las dos posibilidades siguientes:

- Elegir un P_i, obtener todas las tuplas que satisfacen P_i, y verificar cuáles de ellas satisfacen los demás predicados.
- Hacer un barrido de la relación completa verificando cuáles tuplas satisfacen todos los predicados.

Calculamos el costo de cada posibilidad tomando en cuenta la presencia de índices y si se trata o no de índices de agrupamiento, y finalmente elegimos como plan de acceso aquella que resulte de menor costo.

7.4. MÉTODOS DE CÁLCULO DE JUNTAS

La junta es probablemente la operación binaria más frecuente y su implementación ha sido cuidadosamente estudiada. En esta sección consideraremos varias estrategias alternativas para el cálculo de una junta natural: iteración ingenua, iteración por bloques, *sort-merge*, uso de índices, *hash*. Analizaremos también métodos aplicables a juntas múltiples y el índice de junta de mapa de bits.

Iteración Ingenua

Consideremos como ejemplo la junta natural

Hospital |×| *Personal*

y supongamos inicialmente que no disponemos de ningún índice. Digamos que la relación *Hospital* contiene 50 tuplas, y *Personal* contiene 5000. Supongamos que en cada bloque caben 20 tuplas de *Personal* y 25 de *Hospital*. Asumamos que ambas relaciones se hallan almacenadas físicamente en forma contigua, de modo que *Personal* ocupa 250 bloques y *Hospital* ocupa 2.

El método de iteración ingenua consiste simplemente en examinar todos los pares de tuplas posibles formados por una tupla de *Hospital* y una de *Personal*. Esto requiere examinar 5000 × 50 = 250.000 pares de tuplas. ¿Cuántos accesos a memoria secundaria serán necesarios? Esto depende de la estrategia que se siga. Por ejemplo, supongamos que leemos un bloque de la relación *Personal*, y para cada tupla que aparece en este bloque leemos secuencialmente toda la relación *Hospital*. En este caso, necesitamos 5000/20 = 250 accesos para leer todos los bloques de *Personal*. Una lectura completa de *Hospital* requiere 2 accesos; como la cantidad de lecturas completas de *Hospital* es $n_{personal}$ = 5000, el total de accesos a *Hospital* es de 10 000, y el costo total es 10.250 accesos.

$m_H = 50$ $m_P = 5000$

HB cant de H en bloque = 25

PB = 20

Iteración por Bloques

Considerando el método de la sección anterior, es fácil observar que la siguiente modificación será ventajosa. Leemos un bloque de *Personal*; para este bloque de *Personal*, leemos todos los bloques de *Hospital* y hallamos *todas* las tuplas que participan en la junta en este par de bloques. De este modo, la cantidad de lecturas completas de *Hospital* es igual a la cantidad de *bloques* en *Personal*, no a la cantidad de tuplas; el costo total resulta entonces de (5000/20) × 3 = 750 accesos.

Estos cálculos suponen que sólo tenemos espacio en memoria principal para mantener un bloque de cada relación; en la práctica, probablemente dispondremos de más espacio y podremos por ejemplo cargar la relación *Hospital*, que tiene sólo 2 bloques, totalmente en memoria principal. Si hacemos esto y luego leemos la relación *Personal* bloque por bloque secuencialmente, el costo se reduce a 2 accesos para la lectura inicial de *Hospital*, más 250 accesos para la única lectura de *Personal*, con un total de 252 accesos; una mejora notable sobre los 10 250 de la iteración ingenua. Intuitivamente, observamos que este método debe ser óptimo, ya que su costo es simplemente el de leer completamente ambas relaciones, y todo método debe al menos incurrir este costo. En caso de que ninguna de las dos relaciones pueda

ser cargada completamente en memoria principal, podemos modificar el método de modo que cargue tantos bloques como sea posible de una de las relaciones y, para cada conjunto de bloques de la primera relación, recorra completamente la segunda.

Método de *Sort-merge*

En caso que ninguna de las dos relaciones quepa completamente en memoria principal, otra estrategia eficiente consiste en ordenar las dos relaciones de acuerdo con el valor del atributo de junta *(sort)* y luego calcular la junta mediante un simple barrido secuencial de ambas relaciones *(merge)*. Una vez ordenadas las relaciones, el costo del *merge* es simplemente la suma de la cantidad de bloques ocupada por cada relación (252 en el ejemplo); o sea que este método iguala al que almacena una relación totalmente en memoria principal en el caso en el que las relaciones ya están ordenadas de antemano. Si no es así, el costo adicional del ordenamiento es del orden de $B \log B$ para una relación que ocupa B bloques.

k examinar

5000 × 50 = 250000 tupla

$m_P \times m_H$

Uso de Índices

Supongamos ahora que tenemos un índice sobre el atributo *hosp* de la relación *Personal*. Volviendo a la iteración ingenua, podemos ahora modificarla del siguiente modo. Para cada bloque que leemos de *Hospital*, usamos el índice para extraer de *Personal* aquellos bloques que contienen tuplas con los valores de *hosp* que aparecen en este bloque de *Hospital*.

Aunque en este ejemplo este método resultará más costoso que la iteración por bloques, debido al costo extra de los accesos al índice, hay dos situaciones en las cuales su performance es muy superior. La primera es el caso en que las tuplas de *Personal* no se hallan almacenadas en forma contigua en bloques; por ejemplo, si aparecen entrelazadas con las tuplas de otra relación usando el método de agrupamiento. En este caso el uso del índice nos permite acceder sólo aquellos bloques que contienen tuplas de *Personal* ignorando los otros. La segunda situación ocurre si sólo una fracción de las tuplas de la segunda relación participan en la junta. Por ejemplo, supongamos que la relación *Personal* contiene información sobre personal de los hospitales de todo el país, mientras que *Hospital* solamente lista los hospitales de una ciudad dada. En este caso el índice nos permite acceder solamente los

bloques relevantes a la consulta e ignorar los demás. Las ventajas del uso de índices en estas situaciones son tan significativas que generalmente los sistemas consideran la posibilidad de crear un índice expresamente para el cálculo de una junta, si éste no existiera de antemano.

Método *Hash*

Se trata, en realidad, de una familia de métodos que se caracterizan porque es posible acotar la cantidad de comparaciones entre las tuplas de las relaciones intervinientes. En estos métodos no se requiere que las relaciones se encuentren o sean clasificadas, que posean algún tipo de índice o se encuentren limitadas por su tamaño. La idea es separar las tuplas de una relación que pueden ser juntadas con ciertas otras tuplas de la otra relación.

Un primer método a analizar es el llamado *Método Simple de Junta Hash*. Se distinguen dos fases bien definidas. La primera se denomina fase *constructiva* y consiste en armar una tabla hash en memoria con los datos de la primera relación, supuestamente la más pequeña, que llamamos R. Para esto se utiliza una función de hash h que se aplica a cada tupla, específicamente a los valores de los atributos sobre los que se realiza la junta. Asumiremos que las funciones de hash mapean los valores de clave a un espacio de direcciones de manera uniforme. Llamemos X al conjunto de atributos en función de los cuales se realiza la junta. La función h transforma valores de X en direcciones dentro de la tabla hash que designaremos con TH. En TH se pueden almacenar las tuplas completas o simplemente los *tid* (identificadores de tupla) y los correspondientes valores de X. Obviamente, se debe considerar algún método para la resolución de colisiones. En la segunda fase, denominada *exploratoria*, se aplica la función h a las tuplas de la otra tabla, que llamamos S. De esta manera es posible ubicar, en la tabla TH, y juntar con cada tupla de S, las correspondientes filas de R. En cuanto al costo de este método éste resulta del orden de la suma de la cantidad de bloques de ambas relaciones, ya que cada una de ellas es leída una sola vez. Por ejemplo, para realizar la junta entre las tablas *Personal* y *Hospital* se requieren 252 accesos para su lectura. Obviamente, la tabla hash en memoria TH se construiría con la relación *Hospital*, lo que minimizaría las comparaciones para realizar la junta.

Si las relaciones no caben en memoria principal, el Método GRACE, basado en el algoritmo de la máquina de base de datos del proyecto japonés de Quinta Generación, se presenta como una alternativa eficiente.

GRACE se lleva a cabo en dos etapas, para lo cual usa dos funciones de hash. La primer función, que se denomina *función de particionamiento*, la designamos con h_1 y se utiliza para particionar las relaciones R y S en subconjuntos disjuntos. Si M es la cantidad de particiones seleccionada, h_1 aplicado a los componentes de X de cualquier tupla produce un valor entre 0 y M − 1. En la segunda etapa se juntan las correspondientes particiones utilizando el Método Simple anterior. El algoritmo siguiente usa M buffers para manipular las tuplas procesadas durante la primer etapa:

Etapa 1

i. Leer R. Para cada tupla t de R calcular $h_1(t[X])$, lo que resulta en un número de buffer. Grabar la tupla en el buffer en memoria y cuando alguno se complete grabarlo al disco. Después de leer toda R grabar el contenido de los buffers al disco. De esta manera R queda particionada en subconjuntos R_i, con $0 \le i \le M - 1$.

ii. Leer S y hacer lo mismo que en el paso 1, generando una partición para S de subconjuntos S_j, con $0 \le j \le M - 1$.

Etapa 2

Para i = 0, ..., M − 1

iii. Leer R_i y construir con sus tuplas una tabla hash TH en memoria, utilizando una función de hash h.

iv. Leer S_i y aplicar a cada una de sus tuplas la misma función h del paso iii. Para cada tupla de Si, verificar si en TH existe alguna tupla juntable. Si es así, emitir una tupla del resultado y luego continuar con la siguiente tupla de S_i.

En este método, la cantidad de bloques a leer se duplica con respecto al Método Simple; pero todavía resulta más económico que el Sort-Merge analizado anteriormente.

7.5. JUNTA DE MÚLTIPLES RELACIONES

Una manera de resolver una consulta que involucra la junta de múltiples relaciones $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k$, con $k > 2$, es traducir la expresión a una se-

cuencia de juntas binarias. Cada junta, por su parte, puede ser implementada usando diversos algoritmos, tales como los descriptos anteriormente. Existen diversas alternativas de evaluación de la junta múltiple en función de las distintas formas de aplicación de paréntesis. Por ejemplo, si k = 4, tenemos $(R_1 |\times| R_2) |\times| (R_3 |\times| R_4)$, $(R_1 |\times| (R_2 |\times| R_3)) |\times| R_4$, $(R_1 |\times| R_3) |\times| (R_2 |\times| R_4)$, etc., expresiones que, en virtud de las propiedades conmutativa y asociativa de la junta natural, sabemos que son equivalentes.

Para evitar grabar, y luego volver a leer, los resultados intermedios de las juntas, se utiliza una técnica conocida como *pipeline*. De acuerdo con esta técnica, los resultados intermedios de una operación son utilizados inmediatamente en la siguiente junta. De esta manera, sólo el resultado final es grabado al disco. Obviamente, de todas las expresiones equivalentes, algunas son más susceptibles que otras de beneficiarse con la aplicación de la estrategia de pipeline. En general, son las que se encuentran anidadas en paréntesis de izquierda a derecha, por ejemplo las del tipo $R_1 |\times| R_2) |\times| R_3) |\times| R_4$.

A continuación describiremos el algoritmo que implementa la estrategia de pipeline, para k = 4:

```
for each tupla t en R1
    recuperar tuplas en R2 que sean juntables con t en Q1
    juntar t con cada tupla recuperada de R2 obteniendo T1 = t . Q1
    for each tupla t1 en T1
        recuperar tuplas en R3 que sean juntables con t1 en Q2
        juntar t1 con cada tupla recuperada de R3 obteniendo T2 = t1 . Q2
    for each tupla t2 en T2
        recuperar tuplas en R4 que sean juntables con t2 en Q3
        juntar t2 con cada tupla recuperada de R4 obteniendo T3 = t2 . Q3
        agregar T3 al resultado final
    endfor
  endfor
endfor
```


7.6. MÉTODO DE ÍNDICE DE JUNTA CON MAPAS DE BITS

Este método realiza la junta por medio de índices que utilizan como representación vectores de bits comprimidos. Permite juntar múltiples tablas, lo que resulta de aplicación, particularmente, en ambientes de soporte de decisiones (Data Warehousing).

El método resulta de la combinación de las siguientes técnicas:

i) *Índices de Junta*

Un índice de junta, para relaciones *R* y *S*, se define como una relación binaria que identifica cada tupla resultado de la junta, por medio de punteros a las respectivas tuplas de las relaciones *R* y *S*. Se puede considerar entonces un índice de junta como una forma especial de vista materializada.

Cada puntero puede representarse como un valor de una clave, una dirección física, o cualquier otro valor que identifique en forma unívoca a cada tupla en una relación.

El algoritmo para ejecutar la junta de *R* con *S* por medio de este método se reduce a:

a) Leer el índice de junta y recuperar los punteros a pares de tuplas juntables (x,y).

b) Concatenar los pares (x,y) hallados, formando tuplas z y almacenarlas en la relación resultado T.

El costo más importante en este método, una vez construido el índice, es el que surge de recuperar las tuplas juntables. Luego, el mejor caso resulta cuando ambas relaciones, *R* y *S*, están ordenadas o agrupadas en función de los atributos en base a los cuales se realiza la junta. El costo resulta del orden de la suma de la cantidad de bloques que ocupan *R* y *S*. En el peor caso, el orden asciende a la suma de la cantidad de tuplas de *R* y *S*.

ii) *Índices de mapas de bits*

Sea una relación *R* con *N* tuplas o filas y un determinado atributo o columna *A*, tal que |dom(*A*)| (el cardinal del dominio de *A*) sea *M*. Un índice de mapa de bits es un conjunto de *M* vectores o cadenas binarias de longitud *N*. Cada fila de la tabla se asocia con el mismo número de bit en cada cadena. En particular, en la cadena que corresponde al valor que la fila tiene en la columna *A*, ese bit es 1 y 0 en las restantes. Así, por ejem-

plo, si se trata de una relación con datos de personas con 1000 filas y el atributo sobre el que se define el índice es Sexo, el índice de mapa de bits estará integrado por dos vectores o cadenas. Una cadena se utilizará para el valor "Varón" y otra para el valor "Mujer". El primer bit de la cadena correspondiente a "Varón" será 1 si la primera fila de la relación corresponde a un varón y, en ese caso, el primer bit de la cadena correspondiente a 'Mujer' será 0. Veamos otro ejemplo: se registran las ventas de los productos de una empresa en las distintas sucursales. Un índice de mapa de bits tendrá tantas cadenas de bits como sucursales tenga la empresa. Si la fila 331 de la relación de ventas corresponde a un producto que se vendió en la sucursal 14, luego la cadena 14 tendrá un 1 en la posición 331. Las demás cadenas, correspondientes a las otras sucursales, tendrán 0 en la posición 331.

Existen diversas técnicas para compresión de datos, las que permiten reducir drásticamente el espacio necesario para almacenar estos índices. Una ventaja del uso de los índices de mapa de bits radica en la reducción de las entradas/salidas al dispositivo de almacenamiento. Otra ventaja se debe a que las combinaciones de predicados AND, OR y NOT se pueden ejecutar muy eficientemente, en paralelo utilizando las instrucciones de los procesadores de 64 bits o más.

Un caso particular es el método denominado junta estrella. Este método permite realizar la junta de una relación central denominada *tabla de hechos*, con múltiples relaciones descriptivas, denominadas *tablas de dimensiones*. Las claves primarias de las tablas de dimensiones son las claves foráneas de la tabla de hechos y, consecuentemente, las juntas se realizan sobre esos atributos. Para cada posible junta se define entonces un índice de junta en el que se incluyen, para cada fila en una tabla de dimensiones, una cadena de bits comprimida. Esta última representa el conjunto de filas en la tabla de hechos que se juntan con la correspondiente fila en la tabla de dimensiones.

EJERCICIOS

E1 Sean las siguientes tres relaciones:

$R(A, B, C)$, $S(B, C, D)$ y $T(D, E)$

Asumir que:

$n_R = 10.000$, $n_S = 20.000$ y $n_T = 30.000$
$V(A, R) = 100$, $V(B, R) = 200$, $V(C, R) = 200$,
$V(B, S) = 400$, $V(C, S) = 500$, $V(D, S) = 200$,
$V(D, T) = 100$, $V(E, T) = 200$
Un bloque puede alojar 50 tuplas de cualquier relación.
Se pide estimar el tamaño de la junta $R \bowtie S \bowtie T$.

E2. Dada la relación $R(A, B, C)$ y la consulta

```
select *
from R
where A > 10 and B = 10 and C = 5;
```

Asumir que:
$n_R = 10.000$, $b_R = 1000$ (cantidad de bloques de r)
$V(A, R) = 90$, $V(B, R) = 150$, $V(C, R) = 150$.

Suponer que existe un índice de agrupamiento para A e índices de no agrupamiento para B y C.
¿Cuáles de las siguientes opciones requieren la lectura de la mínima cantidad de bloques para la consulta?:

a) leer la relación R íntegramente a memoria.
b) realizar primero la selección A > 10
c) realizar primero la selección B = 10
d) realizar primero la selección C = 5

E3 Sea la siguiente base de datos:

Empleado (nro_e(4), nom_e(32), domicilio(24), ciudad(10), edad(2), dni(8), tel(12), sueldo(6))

Trabaja_en(nro_e, nro_depto)
Departamento(nro_depto(4), nom_depto(20), ubicación(20), ciudad(10))
Gerencia(nro_e, nro_depto, desde(8), hasta(8))

Los valores entre paréntesis indican la longitud en bytes de los atributos. Realizar la optimización algebraica de las siguientes consultas SQL:

i. select e.nro_e, e.nom_e, t.nom_depto, g.nro_e
from Empleado e, Trabaja_en t, Gerencia g, Departamento d
where e.nro_e = t.nro_e and d.nro_depto = t.nro_depto
and g.nro_depto = d.nro_depto and e.ciudad = "Rosario
and e.edad < 35;

ii. select e.nro_e, e.nom_e, e1.nro_e, e1.nom_e
from Empleado e, Trabaja_en t, Gerencia g,
Departamento d, Empleado e1
where e.nro_e = t.nro_e and d.nro_depto = t.nro_depto
and g.nro_depto = d.nro_depto and g.nro_e = e1.nro_e
and desde > 1/1/1996 and hasta < 1/1/1999

E4. Definir la mejor estrategia de evaluación y calcular su costo sobre la base de datos del E3 para las siguientes consultas.

Asumir que:

$n_{Empleados}$ = 100.000 tuplas, tamaño de bloque = 4000 bytes
V(ciudad, Empleado) = 100, V(edad, Empleado) = 35,
V(sueldo, Empleado) = 50.

i. select *
from Empleado
where ciudad = "Córdoba" and edad = 45;

Suponer que existe:

a) Índice de agrupamiento sobre *ciudad*, índice de no agrupamiento sobre *edad*.
b) Índice de no agrupamiento sobre *edad*.
c) Índice de agrupamiento sobre *edad*

ii. select *
from Empleado
where ciudad = "Córdoba" or edad = 45;

Suponer que existe:

a) Índice de agrupamiento sobre *ciudad*, índice de no agrupamiento sobre *edad*.
b) Índice de no agrupamiento sobre *edad*.
c) Índice de agrupamiento sobre *edad*

iii. select *
from Empleado
where (ciudad = "Córdoba" or edad = 45) and sueldo > 1500;

Suponer que existe:

a) Índice de agrupamiento sobre *ciudad*, índice de no agrupamiento sobre *edad*, índice de no agrupamiento sobre *sueldo*.
b) Índice de no agrupamiento sobre *edad*, índice de agrupamiento sobre *sueldo*.
c) Índice de agrupamiento sobre *edad*, índice de no agrupamiento sobre *sueldo*

E5. Definir la mejor estrategia de evaluación y calcular su costo sobre la base de datos del E3 para las siguientes consultas.

Asumir que, en adición a los datos de E4:
$n_{Trabaja_en}$ = 130.000, $n_{Gerencia}$ = 200. Asumir valores razonables para los datos faltantes necesarios para responder a cada consulta.

i. select e.nro_e, g.nro_e
from Trabaja_en t, Gerencia g
where t.nro_depto = g.nro_depto;

Suponer que:

a) No hay índices sobre los atributos involucrados en la consulta.
b) Existe un índice de no agrupamiento sobre g.*nro_depto*.
c) Existe un índice de agrupamiento sobre g.*nro_depto*.
d) Existe un índice de no agrupamiento sobre t.*nro_depto*.

ii. Ídem ejercicio anterior, pero indicando al menos 3 alternativas para ejecutar la siguiente consulta, calculando el costo de cada una de ellas. Suponer los datos necesarios, índices, etc.

```
select e.nro_e, e.nom_e
from Trabaja_en t, Gerencia g, Empleado e
where t.nro_depto = g.nro_depto and e.nro_e = t.nro_e;
```

iii. Ídem anterior para la consulta:

```
select e.nro_e, e.nom_e
from Trabaja_en t, Gerencia g, Empleado e
where    t.nro_depto = g.nro_depto and e.nro_e = t.nro_e
     and e.edad < 30 and e.ciudad = "Rosario";
```

E6. Sean R y S dos relaciones tales que n_R = 2.000.000 y n_S = 1.500.000. En un bloque se pueden almacenar 40 tuplas de R o de S. b_R = 50.000 y b_S = 37.500. Se pide seleccionar la mejor estrategia de junta, entre todas las que conozca y estimar la cantidad de accesos al disco, para cada uno de los casos siguientes de disponibilidad de memoria volátil:

i. 2501 bloques.
ii. 501 bloques
iii. 301 bloques
iv. 101 bloques.

E7. Al describir la junta sobre *múltiples relaciones* aplicamos la estrategia de *pipeline* sobre el método de junta por iteración. Realizar las modificaciones necesarias para aplicar pipeline sobre el método *sort-merge*. ¿Es posible utilizar el método de junta hash con pipeline?

NOTAS BIBLIOGRÁFICAS

Un artículo de Goetz Graefe [110] pasa revista a diversos métodos de optimización de consultas y sirve de punto de entrada a la literatura. Un libro reciente es G. Graefe [111]. Una colección de artículos sobre el tema es la editada por W. Kim, D. S. Reiner y D. S. Batory [138].

Entre los textos generales de bases de datos merece citarse el tratamiento que del procesamiento de consultas hacen P. O'Neil [167] y R. Ramakrishnan y J. Gehrke [174].

CAPÍTULO 8

CONTROL DEL PARALELISMO Y RECUPERACIÓN

8.1. INTRODUCCIÓN

Los problemas conceptuales encontrados en el diseño de sistemas operativos de multiprogramación demostraron tempranamente el cuidado que requiere el manejo de múltiples procesos operando en paralelo. La literatura de sistemas operativos contiene abundante información sobre técnicas para solucionar problemas tales como sincronización y exclusión mutua entre procesos. Sin embargo, los diseñadores de sistemas de bases de datos que permiten a múltiples procesos acceder a los datos en paralelo han debido construir un nuevo conjunto de técnicas ante los nuevos problemas planteados en este contexto.

Estos problemas se caracterizan por la generación de resultados inconsistentes que, como consecuencia de fallas que interrumpen el procesamiento, permiten el almacenamiento de resultados parciales; errores producidos por la interferencia entre procesos en paralelo o concurrentes, y la imposibilidad de conocer con certeza cuándo los datos que graba una aplicación se registran efectivamente en la base de datos en memoria estable.

Tanto el control del paralelismo como la recuperación serán analizados en un ambiente de base de datos centralizado. Expresamente no consideraremos el caso de una base de datos distribuida.

EJEMPLO 8.1: Supongamos que dos procesos, P_1 y P_2, desean modificar la base de datos. A los fines de nuestro ejemplo, supongamos que la base de datos solamente consiste de dos elementos, X e Y; por el momento es irrelevan-

te si X e Y son registros tuplas, relaciones, etc. La función de P_1 es simplemente copiar el valor de X a Y, y la función de P_2 es copiar el valor de Y a X. Si ejecutamos primero P_1 y luego P_2, el resultado final es que tanto X como Y reciben el valor original de X, mientras que si ejecutamos primero P_2, el resultado final es que ambos reciben el valor original de Y. En los dos casos X e Y tienen el mismo valor al finalizar la ejecución. Supongamos en cambio que, para agilizar el desempeño del sistema, permitimos que los pasos de P_1 y P_2 se ejecuten en forma *entrelazada*, en la forma indicada en la Figura 8.1. El efecto de esta ejecución de P_1 y P_2 es intercambiar los valores de X e Y; observamos que X e Y no necesariamente tienen el mismo valor

P_1: Leer el valor de X, X_0

P_2: Leer el valor de Y, Y_0

P_1: Escribir X_0 en Y

P_2: Escribir Y_0 en X

Figura 8.1
Transacciones Entrelazadas

:spués de esta ejecución. O sea, el haber entrelazado las ejecuciones de P_1 P_2 hace posibles resultados incorrectos que ninguna ejecución secuencial uede producir.

A fines de discutir los problemas antes planteados, vamos a introducir la ción de *transacción*. Una transacción es una ejecución de un programa de licación que accede a la base de datos. El concepto análogo en el área de temas operativos es el de *proceso*; la principal distinción es que, a diferen- de los procesos, que suelen ser cíclicos y de duración prolongada, una nsacción generalmente es de corta duración. Típicos ejemplos de transac- nes serían una reserva o cancelación de pasaje en un vuelo, un depósito, ro de fondos o transferencia en cuentas bancarias, la inscripción de un vo estudiante en una universidad, etc.

Una transacción consiste en una sucesión de *acciones*. Una acción es un o elemental y atómico de una transacción, es decir, una operación tal que n se hace o no se hace, pero no puede hacerse parcialmente. Además, el

sistema sólo puede ejecutar una acción a la vez. O sea, el entrelazamiento de las transacciones se da a nivel de las acciones. Por el momento podemos considerar como acciones las operaciones de leer el valor de un ítem de la base de datos y actualizar el valor de un ítem. Las operaciones y cómputos que realice una transacción internamente, sin acceder a la base de datos, son irrelevantes para nuestro análisis. Por ejemplo, la transacción P_1 en el ejemplo 8.1 consiste de las dos acciones "Leer X" y "Escribir Y". No hemos definido qué es un *ítem* de la base de datos: esto dependerá de la implementación del controlador de paralelismo. Para fijar ideas, conviene pensar en un ítem como si fuera un registro o una tupla.

En general una transacción contiene dos tipos de operaciones: operaciones de base de datos, tales como *Leer* y *Grabar*, y operaciones de transacción, conocidas como *Abortar* y *Commit*, término este último que no traduciremos con la finalidad de evitar distorsiones semánticas.

En SQL-92 no existe una sentencia especial para indicar el comienzo de una transacción, aunque ha sido incluida en el futuro estándar SQL:1999. En cambio sí existe un comando de terminación, tanto para la finalización normal como para la anormal. En el primer caso se indica con

Exec SQL Commit

mientras que en el segundo se escribe

Exec SQL Rollback

En este segundo caso, la transacción debe abortar ante una situación anormal, por ejemplo la detección de un error irrecuperable, y, mediante el comando Rollback, solicita al sistema que deshaga los cambios a la base de datos realizados hasta el momento.

Una transacción es iniciada implícitamente al ejecutar una sentencia que requiere el contexto de una transacción. Consideraremos entonces una sentencia de *Comienzo de transacción*, sea ésta implícita o explícita. Cada vez que se inicia una transacción el sistema le asigna un identificador único. En adelante es posible identificar cualquier acción de la transacción por medio de ese identificador.

Con *Commit* se indica que la transacción completó su ejecución exitosamente. En consecuencia, los cambios realizados a los datos resultan permanentes en la bd y visibles a otras transacciones que se ejecutan en paralelo.

El hecho de encapsular las operaciones de base de datos dentro de una

transacción, permite aprovechar las *garantías* que proporciona el sgbd. Esas garantías se conocen como *Propiedades ACID* de una transacción.

8.2. PROPIEDADES ACID

ACID es un acrónimo para las cuatro propiedades que un sgbd garantiza a las transacciones que se ejecutan bajo su control, a saber, Atomicidad, Consistencia, aIslamiento (la I proviene del inglés Isolation) y Durabilidad.

A continuación, examinaremos brevemente cada una de esas propiedades y analizaremos cómo se relacionan entre sí.

Atomicidad

Una transacción, como ya se dijo, consiste de un conjunto de acciones. Pero según esta propiedad, la transacción se ejecuta completamente o no se ejecuta en absoluto. En el primer caso la transacción ejecuta "commit", mientras que en el segundo *aborta*. El sgbd proporciona esta visión de *todo-o-nada* garantizando que si la transacción falla por cualquier motivo, deshará los efectos de las actualizaciones que la transacción haya realizado. Como ejemplo, tomemos el caso de una operación de transferencia entre dos cuentas bancarias. La transacción que implementa dicha operación pudo haberse ejecutado hasta la grabación del saldo de la cuenta *desde*, una vez descontado el monto a transferir; pero antes de grabar el saldo de la cuenta *hacia*, ya incrementado, se produce una falla y la transacción aborta. El resultado parcial producido deja la bd en un estado inconsistente. Afortunadamente, nuestro sgbd garantiza la *atomicidad*, por lo que para nosotros, los usuarios, no se ha realizado ningún cambio a la bd y ésta se encuentra exactamente en el estado en que se encontraba antes de iniciarse la transacción fallida.

Consistencia

De acuerdo con esta propiedad, una transacción transforma un estado consistente de la bd en otro estado, también consistente. Un estado consistente de la bd es aquel que satisface todas las restricciones de integridad. Luego, en principio, esta propiedad de las transacciones depende de cómo está realizado el programa de la transacción y esto es responsabilidad del programador de aplicaciones, independientemente de las otras propiedades. En nuestro ejemplo anterior, la condición de consistencia podría ser que la suma de los saldos, ante una operación de transferencia, se mantenga invariante.

AIslamiento

Por esta propiedad, el sgbd da la imagen de que cada transacción se ejecuta en aislamiento, sin interferencia de otras transacciones. Un conjunto de transacciones se dice *aislado* si el efecto de su ejecución en paralelo es exactamente el mismo que si las transacciones se ejecutaran una a una. La finalidad de la ejecución en paralelo es un mejor aprovechamiento de los recursos computacionales. Otro nombre para esta propiedad es *serializabilidad* y será explicada en detalle en la próxima sección.

Durabilidad

Por esta última propiedad, el sgbd garantiza que las actualizaciones a la bd realizadas por transacciones que ejecutaron *commit* serán durables y públicas. Es decir, los cambios serán registrados en la bd en memoria no volátil, típicamente un disco, y serán visibles a otras transacciones, independientemente de cualquier falla subsiguiente que sufra el sistema.

La propiedad de aislamiento es provista por el mecanismo de *control del paralelismo* de un sgbd y será objeto de análisis en las siguientes secciones. La propiedad de consistencia, como ya se dijo, se satisfará si el programa de aplicación es consistente y el sistema garantiza las demás propiedades. Las otras dos propiedades, atomicidad y durabilidad, son provistas por los componentes que manejan la *recuperación* y serán analizadas a partir de la sección 8.8.

En esta presentación nos limitaremos a analizar transacciones simples, por lo que no consideraremos transacciones con subtransacciones.

8.3. EJECUCIÓN DE TRANSACCIONES

Una *ejecución* de un conjunto de transacciones es una sucesión de las acciones *de* las transacciones con la propiedad de que las acciones de cada transacción aparecen en el mismo orden relativo en que aparecen en la transac-

ción. Por ejemplo, la Figura 8.1 muestra una ejecución de las transacciones P_1 y P_2. Intuitivamente una ejecución es un entrelazamiento particular de las acciones de un conjunto de transacciones.

Nuestro primer objetivo va a ser caracterizar aquellas ejecuciones entrelazadas que son *correctas* para lo cual necesitamos primero definir nuestro criterio de correctitud. Nuestra suposición fundamental en este capítulo será que una transacción, tomada aisladamente, es correcta, o sea que dada una base de datos que satisface todos nuestros criterios de consistencia antes de ejecutar la transacción, seguirá satisfaciéndolos después de ejecutada la transacción. Se sigue inmediatamente de esta suposición que la ejecución *serial*, o sea, no entrelazada, de una cantidad arbitraria de transacciones, también será correcta en el mismo sentido. Por lo tanto, vamos a definir como ejecución entrelazada correcta de un conjunto de transacciones a cualquier ejecución *equivalente* a alguna ejecución serial. Nos queda aún por definir cuándo dos ejecuciones de un mismo conjunto de transacciones son equivalentes. Sean E_1, E_2 dos ejecuciones de un conjunto de transacciones $T_1, T_2, \ldots, T_n$. Decimos que E_1 y E_2 son *equivalentes* si

1. Dado el mismo estado inicial de la base de datos, E_1 y E_2 producen el mismo estado final de la misma.
2. Sea T_i una transacción que ejecuta un cierto paso de lectura de un ítem x en E_1 y E_2; el valor leído por este paso de lectura debe ser el mismo en ambas ejecuciones.

EJEMPLO 8.2: Consideremos dos transacciones T_1 y T_2 que transfieren fondos de una cuenta bancaria a otra. Supongamos que T_1 transfiere \$100 de la cuenta A a la cuenta B y T_2 transfiere \$200 de la cuenta B a la cuenta C. Para esto, cada transacción dispone de tres variables locales S_A, S_B, S_C. La notación

$$T_1: S_A \leftarrow A$$

significa que la transacción T_1 lee el valor actual de la cuenta A a su variable privada S_A; esto no afecta el valor de las otras dos variables. En cambio, la notación

$$T_2: B \leftarrow S_B + 100$$

indica que la transacción T_2 escribe en la cuenta B el valor de su variable S_B más 100; esto sí afecta el valor de B que verán las otras transacciones.

La Figura 8.2 muestra tres ejecuciones distintas de estas transacciones. La ejecución E_1 es serial: no hay paralelismo. E_2 y E_3 no son seriales. Supongamos que tanto A como B tienen un saldo inicial de \$500 y C tiene un saldo inicial de \$0; entonces, después de ejecutar E_1, el saldo de A será de \$400, el de B será de \$400 y el de C será de \$200. Si en cambio ejecutamos E_2 con los mismos saldos iniciales, obtenemos los mismos saldos finales. Pero si ejecutamos E_3, el saldo final de la cuenta B no es de \$400 sino de \$600. ¿Qué ocurrió? T_1 y T_2 leyeron el valor original del saldo de la cuenta B, y la modificación de este saldo realizada por T_2 fue ignorada por T_1; esta última simplemente sumó \$100 al valor original del saldo. Esto es lo que se llama el problema de "actualización perdida".

E_1	E_2	E_3
$T_1: S_A \leftarrow A$	$T_1: S_A \leftarrow A$	$T_1: S_A \leftarrow A$
$T_1: A \leftarrow S_A - 100$	$T_2: S_B \leftarrow B$	$T_2: S_B \leftarrow B$
$T_1: S_B \leftarrow B$	$T_1: A \leftarrow S_A - 100$	$T_1: A \leftarrow S_A - 100$
$T_1: B \leftarrow S_B + 100$	$T_2: B \leftarrow S_B - 200$	$T_1: S_B \leftarrow B$
$T_2: S_B \leftarrow B$	$T_1: S_B \leftarrow B$	$T_2: B \leftarrow S_B - 200$
$T_2: B \leftarrow S_B - 200$	$T_2: S_C \leftarrow C$	$T_1: B \leftarrow S_B + 100$
$T_2: S_C \leftarrow C$	$T_2: C \leftarrow S_C + 200$	$T_2: S_C \leftarrow C$
$T_2: C \leftarrow S_C + 200$	$T_1: B \leftarrow S_B + 100$	$T_2: C \leftarrow S_C + 200$

Figura 8.2
Tres Ejecuciones

Cuando una ejecución E de un conjunto de transacciones es equivalente a alguna ejecución serial, decimos que E es *serializable*. Dada nuestra suposición de que las transacciones por sí solas preservan la consistencia de la base de datos, podemos concluir que toda ejecución serializable preserva la consistencia de la base de datos. El objetivo del módulo del sistema que controla la ejecución de transacciones en paralelo, a veces llamado el *planificador (scheduler)*, es asegurar que todas las ejecuciones sean serializables. Este control puede ejercerse a dos niveles distintos. Si se ejerce a nivel del sistema los programadores que escriben transacciones se desentienden de los problemas causados por el paralelismo; si se ejerce a nivel de cada transacción es la responsabilidad de los programadores garantizar que toda ejecución resultará serializable.

8.4. CANDADOS Y EXCLUSIÓN MUTUA

Los ejemplos vistos sugieren que necesitamos algún mecanismo que garantice a una transacción de acceso exclusivo a ciertos ítems de la base de datos. Por ejemplo, si la transacción T_1 en la Figura 8.2 pudiera evitar que T_2 actualizara la base de datos entre el momento en que T_1 lee el valor de la cuenta B y el momento en que T_1 actualiza este valor, la ejecución incorrecta E_3 no podría ocurrir. Tal mecanismo se conoce como *cierre (locking)*. El objeto de un pedido de cierre sobre un ítem por una transacción es la obtención de un *candado (lock)* sobre ese ítem. Si una transacción T_1 pide cerrar un ítem x y x está *libre*, o sea, ninguna otra transacción ha adquirido un candado sobre x, entonces T_1 adquiere el candado sobre x y continúa ejecutando. Si otra transacción T_2 intenta cerrar el mismo ítem x antes de que T_1 *libere* x, T_2 queda bloqueada hasta tanto T_1 libere x.

Denotemos el pedido de una candado sobre un ítem x como LL(x) y la liberación de x como LI(x). Para garantizar exclusión mutua toda transacción debe obedecer las dos reglas siguientes:

1. Una transacción no puede leer ni actualizar un ítem x hasta tanto no haya ejecutado la operación LL(x).
2. Una transacción que intenta obtener una candado sobre un ítem x que ha sido cerrado por otra transacción T debe esperar hasta que T ejecute la operación LI(x).

Volviendo al ejemplo 8.2 vemos que si T_2 ejecuta LL(B) antes de leer B, y no ejecuta *LI(B)* hasta después de haber actualizado B, T_1 se ve obligada a esperar y por lo tanto T_1 lee el valor de B ya actualizado por T_2. Esto implica que la ejecución E_3 no es posible si el sistema obedece las reglas 1 y 2 dadas arriba. Es natural preguntarse si las reglas 1 y 2 son suficientes para garantizar que toda ejecución será correcta, o sea, serializable. El ejemplo siguiente muestra que no es así.

EJEMPLO 8.3: Volvamos al ejemplo 8.1. Tenemos ahora dos transacciones T_1 y T_2 tales que T_1 copia el valor de x a y y T_2 copia el valor de y a x. La ejecución de la Figura 8.3 obedece las reglas 1 y 2 respecto a la adquisición y liberación de candados, pero no es serializable; es básicamente la misma ejecución que en la Figura 8.1.

T_1	T_2
LL(x)	
Leer x	
LI(x)	
—esperar—	LL(y)
—esperar—	Leer y
—esperar—	LI(y)
LL(y)	
Escribir y	
LI(y)	
	LL(x)
	Escribir x
	LI(x)

Figura 8.3
Ejecución no serializable con candados

De ahora en más, vamos a suponer que toda ejecución de un conjunto de transacciones respeta las reglas 1 y 2. ¿Cómo podemos determinar si una ejecución es serializable o no? Antes de atacar este problema, vamos a refinar nuestro modelo de exclusión mutua. Es claro que si dos transacciones van a leer, pero no a modificar, un cierto ítem, no necesitan excluirse mutuamente. En el modelo que hemos introducido hasta el momento, las reglas 1 y 2 no permiten que una transacción lea un ítem mientras éste está cerrado por otra transacción. Para permitir esto, vamos a distinguir entre dos *modos* de cierre: *modo de lectura (read locking)* y *modo de escritura (write locking)*. La regla 1 no se modifica con este nuevo modelo. Para redefinir la regla 2, digamos que un modo de cierre *conflictúa* con otro si uno de ellos es de escritura. La nueva regla 2 es:

2. Una transacción que desea obtener una candado en modo M sobre un ítem x que ha sido cerrado por otra transacción T con un modo que conflictúa con M, debe esperar hasta que T libere el ítem x.

De ahora en más utilizaremos la notación *LL* para indicar un cierre de lectura y *LE* para indicar cierre de escritura.

EJEMPLO 8.4: La Figura 8.4 muestra una ejecución de tres transacciones T_1, T_2 y T_3. Inicialmente, T_1 cierra el ítem A en modo de lectura. Cuando T_3 pide cerrar A en modo de lectura, puede hacerlo, ya que dos candados de lectura no conflictúan entre sí. T_2 en cambio no puede obtener una candado de lectura sobre C una vez que T_3 ha cerrado C en modo de escritura. El lector tal vez se pregunte si la ejecución de la Figura 8.4 es serializable; la respuesta es que sí, puesto que es equivalente a la ejecución serial T_1, T_3, T_2. En la próxima sección daremos un método para determinar la serializabilidad de una ejecución.

8.5. CARACTERIZACIÓN DE EJECUCIONES SERIALIZABLES

Nuestro objetivo en esta sección es presentar un algoritmo que permita decidir, dada una ejecución, si es serializable. ¿Para qué nos servirá este algoritmo? En la práctica, el controlador del paralelismo no va a permitir cualquier ejecución y a posteriori preocuparse de determinar si fue o no serializable. En cambio, el método más común es definir *protocolos de cierre (locking protocols)*. Un protocolo de cierre es un conjunto de reglas sobre el uso de candados tal que, si toda transacción se compromete a seguir estas reglas, el protocolo garantiza que toda ejecución posible es serializable.

T_1	T_2	T_3
LL(A)		
		LL(A)
LE(B)		
LL(A)		
LL(B)		
		LE(C)
		LL(C)
	LL(C)	
		LL(A)
		LE(B)
	LE(A)	
	LL(C)	
	LL(A)	
		LL(B)

Figura 8.4
Ejecución con Dos Modos de Cierre

La caracterización de ejecuciones serializables que vamos a presentar a continuación va a servir como herramienta teórica para demostrar que un protocolo tiene esta propiedad de garantizar serializabilidad.

Dada una ejecución E de un conjunto de transacciones $T_1, \ldots, T_n$, construimos un grafo llamado *grafo de precedencia*. G contiene un nodo por cada transacción T_i. Si T_i ejecuta una operación de cierre de un ítem x en modo M, y T_j ejecuta posteriormente una operación de cierre sobre el mismo ítem x en un modo que conflictúa con x, G contiene un arco que va de T_i a T_j. Intuitivamente, G representa las restricciones que debe satisfacer cualquier ejecución serial si existe, equivalente a E. Por ejemplo, supongamos que en E la transacción T_i cierra x en modo de lectura, libera x, y luego la transacción T_j cierra x en modo de escritura. Si existe alguna ejecución serial S equivalente a E, T_i debe preceder a T_j en S. De lo contrario, el valor de x leído por T_i sería el valor ya modificado por T_j, mientras que en E era el valor antes de ser modificado por T_j y por lo tanto E y S no podrían ser equivalentes.

EJEMPLO 8.5: La Figura 8.5 muestra el grafo de precedencia para la ejecución de la Figura 8.4. El grafo tiene un nodo por cada transacción T_1, T_2, T_3. Como T_1 cierra A en modo de lectura antes de que T_2 haga lo mismo en modo de escritura, existe un arco dirigido de T_1 a T_2. Además, T_1 cierra B en modo de escritura antes de que T_3 haga lo mismo; por lo tanto hay un arco de T_1 a T_3. Finalmente, dado que T_3 cierra C para escritura antes de que T_2 cierre C para lectura, hay un arco de T_3 a T_2.

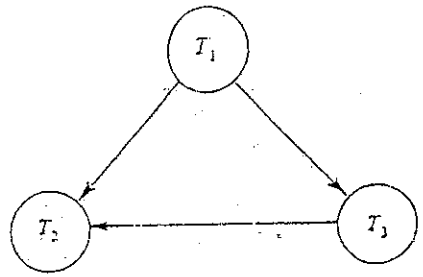


Figura 8.5
Grafo de Predecencia

El siguiente teorema fundamental de la serializabilidad caracteriza las ejecuciones serializables en término de la presencia o ausencia de ciclos en el grafo de precedencia.

TEOREMA 8.1 Una ejecución es serializable si y sólo si su grafo de precedencia es acíclico.

DEMOSTRACIÓN: (Sí) Si el grafo de precedencia es acíclico, podemos aplicar el método de ordenamiento topológico para listar los nodos en orden de manera que todos los arcos queden dirigidos de izquierda a derecha. El orden resultante representa una ejecución serial que satisface todos los requerimientos de precedencia impuestos por el grafo.

(Sólo si) Si el grafo de precedencia es cíclico, ninguna ejecución serial puede satisfacer todos los requerimientos de precedencia dados. Por lo tanto ninguna ejecución serial es equivalente a la ejecución dada, o sea ésta no es serializable.

EJEMPLO 8.6: Ya que el grafo de la Figura 8.5 es acíclico, el teorema nos dice que la ejecución de la Figura 8.4 es serializable. En este caso, el único ordenamiento topológico posible del grafo es T_1, T_3, T_2 y ésta es la ejecución serial equivalente a la de la Figura 8.4. Consideremos en cambio la Figura 8.6, que representa otra posible ejecución del mismo conjunto de transacciones T_1, T_2 y T_3. El grafo correspondiente aparece en la Figura 8.7. Vemos que este grafo tiene un ciclo T_1-T_3-T_2 y por lo tanto la ejecución no es serializable. Intuitivamente, lo que ocurre es que, si existiera una ejecución serial equivalente a la dada, en dicha ejecución serial T_1 tendría que aparecer antes que T_3, para que el valor final de B fuera el mismo; T_3 tendría que aparecer antes que T_2, para que T_2 leyera el mismo valor de C y T_3 leyera el mismo valor de A; y T_2 tendría que aparecer antes que T_1, para que T_1 leyera el mismo valor de A. Esto es imposible, y por lo tanto no existe tal ejecución serial.

T_1	T_2	T_3
		LL(A)
		LE(C)
		LI(C)
		LI(A)
	LL(C)	
	LE(A)	
	LI(C)	
LL(A)		
LE(B)		
LI(A)		
LI(B)		
		LE(B)
		LI(B)

Figura 8.6
Ejecución no serializable

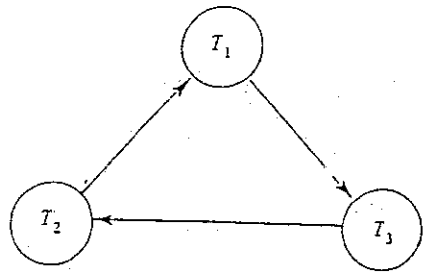


Figura 8.7
Grafo de precedencia cíclico

8.6. Cierre de Dos Fases

El protocolo de cierre llamado *de dos fases* consiste en que ninguna transacción puede hacer una operación de cierre una vez que ha liberado algún ítem que había cerrado. O sea, la evolución de una transacción consiste de una primera fase durante la cual la cantidad de ítems cerrados va creciendo a medida que la transacción adquiere nuevos candados, y una segunda fase durante la cual la cantidad de ítems cerrados va decreciendo a medida que la transacción va liberando ítems cerrados. La segunda fase comienza al realizar la transacción su primera operación LI; de allí en más, el protocolo indica que la transacción no puede volver a hacer ninguna operación LL ni LE sobre ningún ítem.

Ejemplo 8.7: En la Figura 8.3, vemos que ninguna de las transacciones T_1 y T_2 obedece el protocolo de dos fases: T_1 ejecuta LL(y) después de LI(x), y T_2 ejecuta LL(x) después de LI(y). En la Figura 8.4, T_1 y T_2 obedecen el protocolo de dos fases, pero T_3 *cierra B* después de haber liberado *A y C*, violando el protocolo.

El siguiente teorema muestra que este protocolo posee la propiedad que deseamos: si todas las transacciones lo obedecen, tenemos garantizado que toda ejecución será serializable.

Teorema 8.2: Supongamos que todas las transacciones $T_1, \ldots, T_n$ obedecen el protocolo de cierre de dos fases. Toda ejecución de este conjunto de transacciones es serializable.

DEMOSTRACIÓN: Sea E cualquier ejecución de $T_1, \ldots, T_n$. Si E no es serializable, sabemos por el Teorema 8.1 que su grafo de precedencia es cíclico. Sea $T_{i1}, T_{i2}, \ldots, T_{ik}, T_{i1}$ un ciclo en el grafo de precedencia. Por construcción de este grafo, sabemos que T_{i1} ejecutó en E alguna operación de cierre en un modo que conflictuó con una operación de cierre posterior ejecutada por T_{i2}. Por lo tanto, T_{i1} tiene que haber liberado el ítem x antes de que T_{i2} lo haya cerrado.

Por un razonamiento similar, cada T_{ij} tiene que haber liberado algún ítem x_{ij} antes de que $T_{i\,j+1}$ lo cierre, para $1 \le j < k$, y T_{ik} tiene que haber liberado algún ítem z antes de que T_{i1} lo cierre. Pero la conjunción de todas estas aserciones implica que T_{i1} liberó el ítem x antes de cerrar el ítem z, por lo cual T_{i1} no obedece el protocolo de dos fases.

El Teorema 8.2 indica que obedecer el protocolo de dos fases es una condición *suficiente* para que toda ejecución de un conjunto de transacciones sea serializable. En cierto sentido, podemos demostrar que es también condición *necesaria*. Esto no significa que si un conjunto de transacciones no es de dos fases, necesariamente existe una ejecución no serializable de dichas transacciones. Por ejemplo, si todas las transacciones ejecutan solamente cierres de lectura, toda ejecución será serializable independientemente de si las transacciones siguen o no el protocolo de dos fases. Pero sí es posible demostrar que, dado cualquier conjunto de transacciones $T_1, \ldots, T_n$, si al menos una de ellas no es de dos fases, existe una transacción adicional T_{n+1} tal que es posible ejecutar el conjunto $T_1, \ldots, T_{n+1}$ en forma no serializable.

Bloqueos por cierre

La introducción del cierre como método de control del paralelismo trae aparejados dos problemas clásicos ya conocidos en el área de sistemas operativos: *bloqueo (deadlock) y postergación indefinida (livelock o indefinite postponement)*. Consideremos por ejemplo las transacciones de la Figura 8.3. Supongamos que para evitar la ejecución no serializable de la Figura 8.3, T_1 y T_2 modifican su uso del cierre como lo indica la Figura 8.4. Supongamos que T_1 ejecuta la operación de cierre LL(x), e inmediatamente T_2 ejecuta LL(y). Como las operaciones afectan a dos ítems distintos, ambas transacciones siguen ejecutando. Cuando T_1 intenta ejecutar LL(y), debe esperar, porque y está cerrado por T_2. A su vez, T_2 debe esperar que T_1 libere x antes de poder cerrar x. Pero ninguna de las dos transacciones va a liberar el ítem que tiene cerrado hasta tanto la otra haga lo propio, y, por lo tanto, las dos transac-

ciones quedarán bloqueadas indefinidamente hasta tanto el sistema intervenga de alguna forma. Esto es lo que llamaremos *bloqueo*: una situación en la cual dos o más transacciones están condenadas a esperar indefinidamente la liberación de un conjunto de candados.

Para ver un ejemplo de postergación indefinida, supongamos que una transacción T está esperando que otra transacción T_1 libere un ítem x. Apenas T_1 libera x, y antes que T pueda obtener su candado sobre x, otra transacción T_2 se interpone y obtiene un candado sobre x. Si este fenómeno se repite indefinidamente, T nunca podrá continuar su ejecución.

El problema de postergación indefinida es relativamente fácil de evitar; por ejemplo, basta con poner en cola todos los pedidos de cierre del mismo ítem y despacharlos estrictamente en orden de llegada. El problema de bloqueo es algo más delicado. Existen dos tipos de métodos para solucionar el problema de bloqueo: *detección* y *prevención*. En el método de detección el sistema examina periódicamente el estado de las transacciones en ejecución. Si comprueba la existencia de dos o más transacciones bloqueadas, el sistema *aborta* una de ellas, denominada *víctima*, la que debe recomenzar su ejecución. Todos los candados en posesión de la transacción abortada son liberados, permitiendo así que las demás transacciones puedan continuar ejecutando. Hay distintos grados de sofisticación en la comprobación del bloqueo: lo más simple es concluir que existe un bloqueo si hay dos o más transacciones que no han salido del estado de espera durante un tiempo superior a cierto umbral. Un método más cuidadoso consiste en mantener lo que se llama un *grafo de espera*. Este grafo tiene un nodo por cada transacción y un arco dirigido de T_i a T_j si T_i está esperando que T_j libere algún candado. No es difícil demostrar que existe una situación de bloqueo si y sólo si este grafo es cíclico.

8.7. Niveles de Aislamiento

Los protocolos como el de Cierre de Dos Fases garantizan la ejecución serializable de un conjunto de transacciones imponiendo restricciones que degradan el desempeño de la ejecución. Es decir, el grado de concurrencia, medido como la cantidad de transacciones que se ejecutan en paralelo, puede mantenerse bajo a pesar de que recursos tales como la unidad central de proceso se encuentran en estado de espera mucho tiempo. Unas de las causas de esta situación la constituyen los cierres conflictivos a objetos de la bd. Esto ha llevado a imponer cierta flexibilidad a los planificadores (schedulers)

de los sistemas comerciales y se ha visto reflejado en el estándar SQL-92 a través de la definición de los llamados *niveles de aislamiento*.

Los niveles de aislamiento definidos en SQL-92 son los cuatro siguientes: 1) Lectura No Committed (Read UnCommitted), 2) Lectura Committed (Read Committed), 3) Lectura Repetible (Repeatable Read) y 4) Serializable. También se definieron tres subsecuencias de operaciones que pueden producir un comportamiento anómalo, por ejemplo no serializable, denominadas *fenómenos*: Lectura Sucia (Dirty Read), Lectura No Repetible (Non-repeatable Read) y Fantasma (Phantom).

Describiremos en primer término los tres fenómenos mencionados:

F1. Lectura Sucia: Una transacción T_1 realiza una "lectura sucia" si lee un ítem de dato que ha sido previamente actualizado por otra transacción T_2 y, al momento de la lectura por T_1, T_2 todavía no había llegado a ejecutar commit. Si más tarde T_2 aborta, T_1 ha utilizado un valor que nunca existió.

F2. Lectura No Repetible: Una transacción T_1 realiza una "lectura no repetible" si lee un ítem de dato y, cuando intenta leerlo nuevamente, éste ha sido modificado o aun eliminado por otra transacción T_2 la que, a su vez, ya realizó commit.

F3. Fantasma: Una transacción T_1 lee un conjunto de ítems de datos, por ejemplo un conjunto de filas de una tabla, que satisface una determinada condición de búsqueda. Luego, una transacción T_2 crea nuevos ítems, tales como nuevas filas en la tabla, que también satisfacen la condición de búsqueda o modifica ítems con el mismo efecto y ejecuta commit. A continuación si T_1 realiza exactamente la misma consulta, el resultado será distinto del obtenido en la primera oportunidad.

Una transacción que se ejecuta en el primer nivel de aislamiento (Lectura No Committed) exhibe los tres fenómenos descriptos y carece de aislamiento de otras transacciones que se ejecutan concurrentemente. En un sistema que utiliza candados para la sincronización, la transacción que lee no intenta obtener ningún candado y tampoco el sistema verifica que el dato a leer tenga un candado para otra transacción. Este nivel se utiliza en los casos en que las tablas o relaciones son estáticas, cuando la velocidad es más importante que un cien por ciento de exactitud, por ejemplo, en el caso de cálculos estadísticos.

En SQL-92 se declara para este caso

Set Transaction Read Only Isolation Level Read Uncommitted...

En el segundo nivel de aislamiento (Lectura Committed) una transacción exhibirá los fenómenos de Lectura No Repetible y Fantasma, pero no el de Lectura Sucia. Para una transacción en este nivel de aislamiento se declara

Set Transaction Read Only Isolation Level Read Committed.

En el tercer nivel de aislamiento (Repeatable Read) nos aseguramos que la transacción no exhibirá los dos primeros fenómenos, pero todavía puede exhibir el tercer fenómeno (Fantasma). En SQL-92, para una transacción en este nivel, se debe declarar

Set Transaction Read Only Isolati[illegible] Repeatable Read.

Por último, en el [illegible] nivel de aislamiento (Serializable), una transac[illegible] ninguno de los tres fenómenos. Luego, el sgbd garantizará que cualquier ejecución con transacciones en este nivel será totalmente serializable. En SQL-92 se declara, en este caso

Set Transaction Read Write Isolation Level Serializable.

Notemos que en los tres primeros casos, en los que no se garantiza la serializabilidad, sólo se admite declarar las transacciones como de lectura solamente o Read Only.

Cabe mencionar que los sgbd comerciales no necesariamente implementan los niveles de aislamiento propuestos en el estándar, sino que, en la mayoría de los casos han implementado su propia versión.

8.8. RECUPERACIÓN TRANSACCIONAL

La recuperación se define como la actividad de asegurar que las fallas de software y hardware no corrompan los datos persistentes. En un sgbd esta actividad es realizada por un componente denominado Gestor de Recuperación (Recovery Manager).

El Gestor de Recuperación(GR) es el responsable de dos de las propiedades ACID que un sgbd garantiza a toda transacción: *Atomicidad y Durabilidad*. Para esto, el GR asegura que la bd contendrá todos los cambios realizados por las transacciones que hicieron commit (Durabilidad) y ninguno de los cambios de las transacciones que abortaron (Atomicidad).

El GR se ocupa del procesamiento de las operaciones abortar (abort), commit y *reiniciar* (*restart*). La finalidad de reiniciar es restaurar la bd, ante una falla de sistema, a un estado consistente inmediatamente anterior a la falla.

Una transacción es abortada como consecuencia de una *falla*. El GR debe poder recuperar eficientemente a un estado consistente de la bd, ante cualquiera de los siguientes tres tipos de fallas: *de transacción, de sistema y de medio*. El primer tipo ocurre cuando una transacción aborta, por ejemplo, por ella misma al detectar una situación anormal, a causa de un error de programa o por el sgbd al ser seleccionada como víctima en una situación de bloqueo o deadlock.

Como sabemos, existen dos tipos de almacenamiento: i) la memoria volátil o primaria, la que almacena los programas durante su ejecución, así como los datos que éstos utilizan, y ii) la memoria no volátil o *estable*, normalmente un disco, la que almacena la base de datos persistente. Las fallas de sistema son las que afectan de manera impredecible el contenido de la memoria volátil, tales como la interrupción imprevista de la energía, un error de memoria o de procesador, un error del sistema operativo o del sgbd.

Las fallas de medio tienen que ver con la destrucción de alguna parte de la memoria no volátil, por ejemplo, por el "aterrizaje" de las cabezas lectograbadoras sobre la superficie magnetizada de un disco.

En este punto es necesario tener en cuenta lo dicho en el Capítulo 6 sobre manejo de buffers. El área de buffers se denomina también *memoria cache*. El *Gestor de Cache* (GC) o Cache Manager, soporta operaciones llamadas *Fetch(x)* y *Flush(x)*, mediante las cuales lee el bloque que contiene *x* desde el disco (y lo almacena en la cache) y graba al disco la página de cache que contiene *x* respectivamente.

Consideremos la situación en que tenemos una página en memoria cache conteniendo una o más filas de una tabla. Esa página pudo haber sido actualizada por varias transacciones que ejecutaron *grabar* para filas almacenadas en esa página; pero el algoritmo de reemplazo de páginas definió que esa página permaneciera en memoria cache y no fuera a regrabarse al disco. Luego ambos contenidos son evidentemente diferentes. Más aún, las transacciones que actualizaron la página ejecutaron su commit. ¿Qué ocurre ahora si se produce una falla en la que debe reiniciarse el sistema? Sabemos que el contenido de la memoria volátil se pierde irremediablemente y el contenido de la bd en el disco no había sido actualizado. ¿Cómo hace el sgbd para cumplir con su compromiso de hacer que los cambios de transacciones que hicieron commit permanezcan en la bd? Una solución es utilizar un archivo auxiliar, denominado bitácora o *log*, en el que se registran los cambios que producen las transacciones a la bd.

Las técnicas que describiremos en las próximas secciones se aplican a los

dos primeros tipos de fallas mencionados. Las fallas de medio serán tratadas en la sección 8.13.

8.9. GESTIÓN DE *LOGS* O BITÁCORAS

El *log* es un archivo secuencial que contiene todos los datos relativos a las inserciones, eliminaciones y cambios efectuados a cualquier ítem de la bd. Cada vez que una transacción actualiza, por ejemplo, una fila de una tabla el sgbd genera una entrada en el log.

Con toda la información que se almacena habitualmente en el archivo log su utilidad trasciende a la del proceso de recuperación. Otros usos significativos son la auditoría de procesos transaccionales, la evaluación del desempeño del sistema en lo relativo a capacidad de procesamiento y tiempos de respuesta, distribución de costos y facturación a los diversos usuarios, etc.

Un componente del sistema llamado *Gestor de Logs (GL)*, o Log Manager, se ocupa de las operaciones de grabación y lectura del log.

Cada registro del log cuenta con un *encabezado* y un *cuerpo*. El encabezado contiene los siguientes datos:

- Número de Secuencia del registro del Log (LSN); es la clave primaria de este archivo y está integrado por el número de archivo más la dirección relativa en bytes del registro en el archivo.
- LSN del registro previo en el log.
- Marca de Tiempo (Timestamp) con tiempo de creación del registro.
- Identificador de la transacción que grabó el registro (TID).
- Registro previo de esta transacción (LSN).

Esta información es de uso del GL. Los LSN's le permiten recorrer el log en ambos sentidos y recuperar los registros correspondientes a cada transacción de manera eficiente.

El cuerpo de un registro del log puede tener un tamaño variable y contiene toda la información necesaria para la recuperación. El registro completo puede ocupar desde unos pocos bytes a varias páginas.

Los archivos en los que reside el log son duplicados y, habitualmente, las copias son grabadas en distintos dispositivos, con el fin de minimizar la posibilidad de pérdida de datos ante cualquier tipo de falla.

Existen dos formas de acceder en modo lectura al log. Ya que éste es considerado una tabla, puede ser consultado por medio de SQL. La otra forma

de acceder es de bajo nivel y permite recuperar un registro por vez, con acceso directo a través del LSN. En el primer caso, por razones de seguridad, es posible que el acceso se realice usando el mecanismo de vistas de SQL.

Para grabar un registro en el log, el GL lo estructura armando el encabezamiento, mientras el cuerpo se recibe como parámetro. Luego le asigna espacio al final del log y lo graba físicamente. En general, los registros son conservados en buffers en la memoria volátil, por razones de eficiencia. Luego, estos registros pueden perderse si se produce una falla.

Normalmente, el contenido del buffer es grabado a la memoria volátil en el momento en que el buffer se completa; pero existen situaciones en que resulta necesario grabar al log inmediatamente, por ejemplo en el caso de un commit, para lo que usa un comando *Flush_log*. Recordemos que debido a que el log se encuentra duplicado, las grabaciones físicas deben realizarse dos veces. Éste es un proceso costoso en términos de desempeño, por lo que se trata de optimizar la grabación. Para esto, se graban las páginas que se han completado y se trata de grabar tantas páginas como sea posible en una única operación de grabación física, independientemente de la estrategia adoptada (por ejemplo, LRU).

Un aspecto importante es el orden de grabación de los registros: el log debe reflejar el orden en que se realizaron las operaciones de las transacciones, en particular, las operaciones conflictivas. Estas últimas son las realizadas al mismo ítem por dos transacciones concurrentes y donde una de ellas es grabar. El motivo de esto es que, ante una caída del sistema, el GR ejecutará un proceso de *rehacer* que permitirá recuperar los efectos de las operaciones, en el caso de las transacciones que llegaron al commit.

Como parte del log, el GR mantiene en memoria estable listas con identificadores de transacciones activas, transacciones que llegaron al commit y transacciones que abortaron.

Si bien muchos sistemas usan el mismo archivo de log para grabar las imágenes previas y las posteriores, es decir, los valores de los ítems previos y posteriores a la actualización, otros usan logs separados. En este último caso, y debido a que las imágenes previas no resultan necesarias después que la transacción ejecutó commit, ya que no se utilizan para la recuperación, pueden ser borradas en breve tiempo. La desventaja de esta estrategia son las grabaciones extras implicadas por el mantenimiento de los dos logs.

Dependiendo del nivel de actividad, un sgbd puede acumular en el log una gran cantidad de registros que deberán ser procesados en la eventualidad de una falla. Sin embargo, los registros antiguos posiblemente no tendrán incidencia en el estado de la bd, pero insumirán tiempo innecesario ante una recuperación. Para reducir el tamaño del log a procesar, el sgbd graba un registro en el log, llamado *punto de verificación* o *checkpoint*, coincidiendo con la copia a

memoria estable de las imágenes posteriores de las actualizaciones de transacciones que ejecutaron commit y las imágenes previas correspondientes a transacciones abortadas. Daremos más detalle sobre este tópico en la sección 8.12.

Veamos ahora un ejemplo de los datos que se registran en el log ante una ejecución de transacciones.

EJEMPLO 8.8: consideremos la ejecución E_2 de la Figura 8.2 y veamos en la Figura 8.8 qué registros se graban en el log ante cada acción elemental de las transacciones. Obsérvese que Leer no genera ninguna entrada en el log; pero la primera operación de la transacción genera un registro de comienzo.

Operación	Entrada en el log	
T_1: Leer A 700	(T_1 Comienzo)	No se registra la lectura, pero por ser la primer acción de T_1 se toma como Comienzo
T_2: Leer B 850	(T_2 Comienzo)	Ídem anterior
T_1: Grabar A, 600	(T_1, A 700, 600)	Este registro representa la actualización de A en el log. 700 y 600 son los valores de las imágenes previa y posterior, respectivamente.
T_2: Grabar B, 600	(T_2, B, 850 600)	Ídem anterior para B. 850 y 600 son los valores de las imágenes previa y posterior.
T_1: Leer B 600	no se genera entrada en el log	
T_2: Leer C, 500	no se genera entrada en el log	
T_1: Grabar C 700	(T_1 C 500 700)	Actualización de C por T_1 con 500 y 700 los valores de las imágenes previa y posterior.
T_2: Abortar	(T_2, Abortar)	Este registro invoca la ejecución de deshacer.
T_1: Commit	(T_1, Commit)	Entrada correspondiente al commit de T_1. Graba el buffer del log en memoria estable.

Figura 8.8

8.10. EL GESTOR DE RECUPERACIÓN

Como se recordará, el Gestor de Recuperación (GR) es el responsable del procesamiento de las operaciones abortar, commit y reiniciar. A continuación describiremos los efectos de cada una de esas operaciones.

- Abortar transacción restaura los valores que tenía un ítem de dato previo a la ejecución de la transacción. Abortar es irrevocable, en el sentido de que una transacción abortada nunca podrá hacer commit.
- Commit transacción deja en firme las actualizaciones realizadas por la transacción en la bd. Commit también es irrevocable: una transacción que ejecutó commit no puede ser luego abortada.
- Reiniciar ejecuta abortar para las transacciones que estaban activas al momento de producirse la falla que motivó su ejecución; graba en la memoria estable las actualizaciones realizadas por transacciones que ejecutaron commit y deshace los cambios de las transacciones abortadas.

Asumiremos que las aplicaciones declaran transacciones encerrando las acciones elementales dentro de un par *Comienzo-Commit* o *Comienzo-Abortar*. Al ejecutarse Comienzo el sgbd le asigna a la transacción un identificador único T_i.

El GR debe respetar una serie de reglas o protocolos tales como **Grabar Antes en el Log (WAL – Write Ahead Log)** y **Forzar al Log en el Commit (FLC – Force Log at Commit)**.

Durante el procesamiento normal, cualquier actualización a un ítem de dato genera dos operaciones: una grabación en la copia de la página del ítem que reside en la memoria cache y otra en el buffer del log. En algún momento, no predecible, el algoritmo de reemplazo de páginas puede decidir regrabar esa página en memoria cache a la bd en el disco. Si el GC grabara el contenido del buffer del log al log en memoria estable *después* de regrabar la bd, en caso de producirse una falla, la recuperación sería imposible ya que no habría manera de *deshacer* los cambios realizados por la transacción. Con el fin de evitar esta situación el GR debe respetar el protocolo WAL, por el cual los registros del log son grabados físicamente en memoria no volátil previamente a grabar la página afectada en la bd permanente.

El protocolo WAL se puede detallar por medio de las siguientes reglas:

1. Cada página en memoria cache tiene un LSN (Número de Secuencia del Log) correspondiente al registro del log de la última actualización realizada a esa página.
2. Cada actualización debe mantener el LSN de la página.
3. Cuando una página va a ser copiada (Flush) a memoria no volátil, el GC debe solicitar previamente al Gestor de Logs que copie todos los registros del log residentes en los buffers, incluyendo al correspondiente al LSN de la página, al archivo del log en memoria permanente.
4. Finalizada la grabación del log, la versión de memoria cache de la página puede regrabarse sobre la versión anterior de la página en la bd persistente.

También durante el procesamiento normal, podría ocurrir el siguiente caso. Una transacción llega a su punto de commit, o ejecuta commit, pero todas las páginas que modificó permanecen en memoria cache, del mismo modo que los registros del log en el buffer. Si en esta situación se produjera una falla, en el momento de la recuperación no habría ningún elemento que permitiera *rehacer* los efectos de la transacción. Para evitar esto, como parte de la ejecución del commit el GR debe forzar la grabación al log no volátil de todos los registros del log que permanezcan en el buffer. Concretamente el protocolo FLC establece que los registros del log de una transacción deben ser grabados al almacenamiento no volátil como parte del commit de la transacción.

Una transacción se considera que hizo commit en el momento en que su registro de commit se graba en el log en la memoria no volátil.

Reiniciar

Durante la recuperación después de una caída del sistema, el GR realiza la operación de reiniciar. Para esto, en primer término, lee el checkpoint más reciente y busca hacia el final del log para recuperarse a sí mismo, reconstruyendo las estructuras de datos necesarias para su funcionamiento eficiente. Luego inicia la tarea de recuperación propiamente dicha para lo cual aplica alguno de los algoritmos que describiremos en la próxima sección. Por el momento describiremos las operaciones básicas: deshacer (UNDO) y rehacer (REDO).

Deshacer

El GR requiere la operación *deshacer* si permite que las actualizaciones de una transacción que no llegó al commit sean grabados en la bd persistente. En la eventualidad de una falla, la bd contendrá esas actualizaciones indebidas. Éstas deberán ser *deshechas* en la operación reiniciar con la finalidad de que en la bd sólo permanezcan los efectos de las transacciones que hicieron commit.

La regla para *deshacer* se expresa por el protocolo WAL: si un ítem de dato en la bd persistente contiene el último valor actualizado por una transacción que hizo commit, ese valor debe ser resguardado (en el log) antes de ser modificado en la bd persistente por una transacción que no hizo commit.

La operación *deshacer* puede reconstruir un estado anterior de un ítem de la bd a partir del estado actual de ese ítem y registros del log.

Rehacer

El GR requiere la operación *rehacer* si permite que las actualizaciones de una transacción que llegó al commit sean grabadas en la bd persistente después del commit. En la eventualidad de una falla, la bd no contendrá las actualizaciones de algunas transacciones que llegaron al commit. Éstas deberán ser *rehechas* en la operación reiniciar para restablecer la bd a su estado correcto, es decir, incluir el efecto de todas las transacciones que llegaron a ejecutar commit.

La regla para rehacer expresa que, antes que una transacción pueda hacer commit, el nuevo valor para el ítem de dato actualizado debe instalarse en el log.

La operación *rehacer* puede construir el nuevo estado de un ítem de la bd a partir de un estado anterior de ese ítem y registros del log.

Idempotencia de Reiniciar

Una falla puede interrumpir el procesamiento de cualquier operación, incluso de *reiniciar*. Esto impone al GR la restricción que *reiniciar* sea idempotente: cualquier secuencia de ejecuciones incompletas seguidas por una ejecución completa de *reiniciar* debe tener el mismo efecto que una única ejecución completa de *reiniciar*.

8.11. ALGORITMOS DE RECUPERACIÓN

Existen diversos algoritmos de recuperación que utilizan el log. En esta sección describiremos uno de ellos, el Deshacer/Rehacer. No incluiremos en esta descripción los detalles de cada operación ni consideraremos aspectos de eficiencia.

Algoritmo Deshacer/Rehacer

También denominado "Algoritmo de Recuperación con Actualización Inmediata de la Base de Datos", aplica las reglas enunciadas más arriba para rehacer y deshacer. Este algoritmo incorpora al log las tres siguientes listas, las que se graban como un archivo secuencial: transacciones activas, transacciones que hicieron commit y transacciones que abortaron.

A continuación describiremos los distintos tipos de registros del log que son utilizados por este algoritmo:

Actualización: documenta una operación de grabación de una transacción con el siguiente detalle:

- Encabezamiento, ya descripto en la sección 8.9, incluyendo el LSN de este registro, el identificador de la transacción que ejecutó la actualización, el LSN del registro anterior de la misma transacción, etc.
- Nombre o dirección en la bd persistente del ítem de dato actualizado.
- Desplazamiento y longitud de la porción modificada del ítem de dato.
- El valor anterior de la porción modificada del ítem de dato (imagen previa)
- El valor nuevo de la porción modificada del ítem de dato (imagen posterior).

Commit: expresa que la transacción llegó al commit. Contiene el encabezamiento y el tipo de registro.

Abortar: expresa que la transacción abortó. Contiene el encabezamiento y el tipo de registro.

Checkpoint: documenta la realización de un checkpoint con el siguiente detalle:

- Lista de las transacciones activas al momento del checkpoint.
- Lista de los ítems de datos que fueron modificados y permanecían en memoria cache.

La operación de reiniciar procesa el log en dos pasadas. La primera corresponde a Deshacer y procesa los registros del log desde final (el registro más reciente) hasta el primer checkpoint que encuentre o, en su defecto, el comienzo del archivo. La segunda pasada corresponde a Rehacer y procesa los registros desde el último checkpoint grabado, o el comienzo del archivo, hasta el registro más reciente.

Durante la búsqueda hacia atrás, correspondiente a deshacer, *reiniciar* mantiene una lista de transacciones que hicieron commit y una lista de transacciones que abortaron, las que denominaremos Lcom y Labo, respectivamente. Deshacer procede de la siguiente forma con cada registro del log:

- si es un registro de *commit*, agrega el identificador de transacción a Lcom;
- si es un registro de *abortar*, agrega el identificador de transacción a Labo;
- si es un registro de *actualización*:
  - i. si la transacción está en Lcom, ignora el registro;
  - ii. si la transacción no está en Lcom ni en Labo, luego estuvo activa al momento de la falla. Para considerarla abortada la incorpora a Labo;
  - iii. si la transacción está ahora en Labo graba la imagen previa al ítem de dato en memoria cache. Si el ítem no está en cache, lo lee desde la bd persistente, por medio de un fetch. Si la transacción no tiene más registros de actualización para deshacer, es eliminada de Labo.

Rehacer realiza una búsqueda hacia delante en el log y, por cada registro de actualización de una transacción en Lcom, actualiza el ítem de dato correspondiente en memoria cache. Si la página que contiene ese ítem no está en cache, es leída con fetch desde la bd persistente. Cuando rehacer llega al final del log ha reinstalado todas las actualizaciones de transacciones que hicieron commit.

Al finalizar los dos procesos, el estado combinado de la bd persistente y la memoria cache contiene los efectos de todas las transacciones que llegaron al commit y ninguna actualización de las transacciones abortadas.

8.12. *CHECKPOINTS* O PUNTOS DE CONTROL

Durante el proceso completo de reiniciar no es posible ejecutar ningún tipo de transacción, por lo que resulta importante reducir el tiempo de esta tarea. Una forma de optimización la proporcionan los checkpoints.

Cuanto más frecuentemente se ejecute un checkpoint, menor será la longitud del log a procesar por reiniciar, mejorando así los tiempos del proceso. Pero el checkpoint también insume tiempo, por lo que es necesario seleccionar una estrategia adecuada para realizarlo.

Una solución la brinda el checkpoint borroso o *fuzzy checkpoint*, denominado así porque se realiza mientras los objetos están siendo modificados. Para realizar un checkpoint borroso, el GR realiza las siguientes actividades:

i. No acepta ninguna nueva operación de actualización, commit o abortar.
ii. Construye una lista de todas las páginas en memoria cache que han sido modificadas (páginas sucias).
iii. Construye una lista de todas las transacciones activas con los LSN que apuntan al último registro grabado del log, para cada transacción.
iv. Graba un registro de checkpoint al log, el que incluye la lista de transacciones activas.
v. Acepta nuevamente las operaciones de actualización, commit y abortar.
vi. Mientras ejecuta las operaciones del paso v, realiza *flush* a la bd permanente de todas las páginas sucias de la lista construida en el paso ii. Esta tarea es realizada por el Gestor de Cache en la medida que disponga de tiempo libre.

El siguiente checkpoint sólo se puede realizar una vez que el GR terminó con el paso vi, es decir, después que las páginas sucias han sido regrabadas en el disco.

8.13. RECUPERACIÓN DE MEDIO

Hasta el momento hemos asumido que las fallas que pueden ocurrir sólo afectaban a la memoria volátil. Pero, como se mencionó en la sección 8.8, pueden ocurrir fallas de medio que las podemos definir como la pérdida de una parte de la memoria estable o permanente. En este caso, se debe recurrir a una copia de resguardo de la bd, generalmente una cinta o *cartridge*. Con esta copia y el log, normalmente sus copias de respaldo más el log actual, es posible recuperar la bd a un estado actualizado por medio de un proceso similar al que se usa para recuperar en los casos de falla de sistema.

Debido a que la solución recién descripta puede insumir demasiado tiempo para ciertos ambientes de procesamiento en línea, algunos sistemas usan discos espejados (mirrored disks). En este caso, se utilizan dos discos físicos para cada disco lógico, por lo que cada disco tiene una copia de respaldo actualizada al instante.

Una tecnología más general es RAID (Redundant Array of Inexpensive Disks), la que usa varios discos y admite que uno o más de ellos fallen. Esto es menos oneroso que la redundancia total, aunque un sistema RAID es más costoso que discos únicos de la misma capacidad. Existen diversas categorías de RAID, de las cuales mencionamos tres: RAID 0 es "disk stripping" o grabación de datos a través de varias unidades de disco. RAID 1 es almacenamiento cien por ciento redundante. RAID 5 distribuye los datos e información de paridad sobre varios discos, con redundancia suficiente como para alcanzar muy altos niveles de tolerancia a fallas. Se considera que para bases de datos muy grandes estas tecnologías pueden ser muy costosas, pero al menos podrían aplicarse al log.

Copias de Archivo

Como mencionamos antes, la recuperación en caso de falla de medio requiere mantener una copia de respaldo de la bd y una copia del log. Esta última debe contener todas las actualizaciones realizadas por transacciones que se ejecutaron después de realizar la copia de respaldo de la bd y que hicieron commit.

La creación de la copia de la bd consiste simplemente en copia la bd permanente, en su totalidad, a otro medio. Pero esto puede consumir demasiados recursos del sistema. Una alternativa es hacer copias diferenciales: se realiza una copia de la bd íntegra y luego, en lugar de hacer nuevas copias completas, sólo se archivan las páginas que han sido modificadas.

EJERCICIOS

E1. Dada la siguiente ejecución de las transacciones T_1, T_2 y T_3:

$\text{Leer}_2(y)$; $\text{Leer}_3(z)$; $\text{Leer}_1(y)$; $\text{Leer}_1(x)$; $\text{Grabar}_1(x)$; $\text{Leer}_3(x)$; $\text{Leer}_2(x)$; $\text{Grabar}_2(y)$; $\text{Grabar}_3(x)$

Considerar que para cada operación de bd el sistema aplica los correspondientes candados
¿A qué ejecución serial es equivalente? Justifique.

E2. Dados los siguientes planes de ejecución, por medio de los arcos en su grafo de precedencia

i. S_1: $T_1 \rightarrow T_2$, $T_3 \rightarrow T_4$, $T_3 \rightarrow T_1$, $T_4 \rightarrow T_2$, $T_1 \rightarrow T_4$
¿Es S_1 serializable? En caso afirmativo hallar una ejecución serial equivalente a S_1.

ii. S_2: $T_3 \rightarrow T_1$, $T_3 \rightarrow T_4$, $T_1 \rightarrow T_2$, $T_4 \rightarrow T_2$, $T_2 \rightarrow T_3$
¿Es S_2 serializable? En caso afirmativo hallar una ejecución serial equivalente a S_2.

E3. Dado el siguiente plan de ejecución serial de las transacciones T_1, T_2 y T_3

$S = \text{leer}_1(x)$; $\text{grabar}_1(x)$; $\text{leer}_2(x)$; $\text{grabar}_2(y)$; $\text{leer}_3(x)$; $\text{grabar}_3(y)$

Hallar cuántos planes de las seis acciones de S son equivalentes a S y cuáles son esos planes.

E4. Dado el siguiente plan de ejecución de acciones

$\text{leer}_1(x)$; $\text{leer}_2(y)$; $\text{leer}_1(z)$; XXX; $\text{grabar}_1(x)$; $\text{grabar}_2(z)$

¿Por qué acción sustituiría XXX para que la ejecución no sea serializable?

E5. Dar un plan de ejecución serializable, formado por la mínima cantidad de transacciones posible, con un mínimo de acciones de lectura y grabación, y que las transacciones no sigan el protocolo de cierre de dos fases.

E6. Sean las siguientes tres transacciones:

T_1 : leer(o_1); $x \leftarrow o_1$; leer(o_2); $y \leftarrow o_2$; leer(o_3); $z \leftarrow o_3$; $o_4 \leftarrow x+y+z$; grabar(o_4)
T_2 : leer(o_4); $x \leftarrow o_4$; leer(o_2); $y \leftarrow o_2$; $o_4 \leftarrow x+y$; grabar(o_4)
T_3 : leer(o_2); $x \leftarrow o_2$; $o_2 \leftarrow x+1$; grabar(o_2)

i. Encontrar los posibles conflictos y establecer las dependencias que se derivarían de los mismos.
ii. Mostrar todas las secuencias de operaciones entrelazadas de dichas transacciones que no sean serializables.
iii. Indique para las secuencias halladas en el punto anterior, si se trata de casos de actualización perdida, lectura sucia o lectura irrepetible.
iv. Determinar si hay alguna secuencia entrelazada serial de las transacciones que convierta a alguna de las mismas en inútil tomando en cuenta el resultado final.

E7. Considere un protocolo que introduce la aplicación de candados compartidos y exclusivos sobre los objetos, y que se define por las siguientes reglas:

- Toda lectura/escritura sobre un objeto debe estar precedida por la aplicación de un candado compartido/exclusivo sobre el mismo.
- Los candados de escritura se liberan al final de la transacción.
- Los candados de lectura se liberan a continuación de la operación de lectura.

i. Para el caso del E5, ¿hay alguna secuencia entrelazada de operaciones que, respetando las restricciones dadas por la semántica de los candados, no resulte serializable?

ii. Para el caso anterior, ¿hay alguna secuencia que lleve a una situación de bloqueo por cierres?

iii ¿Qué diferencia observa en el uso de este protocolo respecto del protocolo de dos fases?

E8. Se tiene una transacción T_1 que se ocupa de leer los saldos de todas las cuentas de un banco y sumarizarlas, y otra T_2 que realiza una transferencia de un monto determinado de una cuenta a otra, debitando el monto de la primera y acreditando el monto a la segunda. Si se asume un total de tres cuentas, se pide:

i. Mostrar todas las secuencias entrelazadas de las operaciones de dichas transacciones que no sean serializables.

ii Determinar cuáles de las secuencias halladas en el punto anterior constituyen casos de actualización perdida, lectura sucia o lectura irrepetible. ¿Hay algún otro caso no comprendido en estas categorías?

iii Mostrar que la aplicación del protocolo del ejercicio anterior admite mayor cantidad de secuencias entrelazadas de operaciones de T_1 y T_2 que el protocolo de dos fases.

iv. Bajo iii, ¿hay alguna ejecución no serializable?

v. Bajo iii, ¿hay posibilidad de bloqueo por cierres?

E9 Se tiene una transacción T_1 que realiza lo siguiente:

```
T1:     v := 0;
        Para cada miembro e de la colección C realizar
        v := v + e valor; / valor es un entero
        o1 := v;
        grabar(o1);   / o1 es un objeto independiente de C
```

Si una transacción T_2 agrega un elemento nuevo a la colección C, mostrar que cualquier orden en que se realice el barrido de C por parte de T_1, si éste se efectúa en forma concurrente con T_2, hay una ejecución entrelazada de operaciones que no es serializable.

E10. En una base de datos existe una relación *CUENTAS* con atributos *Nro* (el número de la cuenta) que es clave y *Saldo* (el saldo de la cuenta, admitiéndose valores negativos). Se cuenta, a su vez, con dos programas distintos, llamados A y I que realizan lo siguiente:

A: accede a la tabla CUENTAS, tupla por tupla, en secuencia ascendente de NRO, y calcula la suma de los saldos.

I: recibe del usuario 3 números de cuenta N, M y K y un valor positivo D, procediendo de la siguiente forma:
si la tupla con NRO = K tiene un valor de SALDO mayor o igual que D, resta D al SALDO de la tupla con NRO = N y suma D al SALDO de la tupla con NRO = M.
En otro caso deja la bd sin modificar.

Suponer que se invoca una ejecución A1 de A y al mismo tiempo otro usuario invoca una ejecución I1 de I con N = K = 1, M = 2 y D = 50, y luego otra ejecución I2 de I, con K = 2, N = 574, M = 575 y D = 50. Si la instancia de CUENTAS al comenzar la ejecución A1 paralela a I1, y I2 a continuación de I1 era: (1,100), (2,0), ... otras tuplas ..., (574,100), (575, 0).

i Considerando que cada ejecución de I responde al protocolo de dos fases para sus candados, y A no realiza cierre alguno, encontrar una ejecución entrelazada de operaciones de lectura y grabación para A1, I1 y I2 de modo que sus efectos sobre la relación CUENTAS y la sumatoria calculada, coincidan con los efectos de cada una de las ejecuciones seriales posibles.

ii. Dentro de lo considerado en el punto i, ¿existe alguna ejecución entrelazada cuyos efectos no coincidan con los de ninguna de las ejecuciones seriales posibles?

E11 Sea la siguiente secuencia de acciones correspondientes a las transacciones T_1, T_2, T_3 y T_4:

T_1: Leer A, T_1: Leer B, T_1: A:=A+5, T_1: Grabar A, T_2: Leer A, T_1: Grabar B, T_1: Commit, T_2: Leer C, T_2: C:=C+A*10, T_3: Leer A, T_2: Grabar C, T_2: Abortar,

T_3: Leer B T_3: B:=B+A T_4: Leer A, T_3: Grabar B, T_3: Commit
T_4: A:=A*100 T_4: Grabar A *¡se produce una falla!*

i. Mostrar la secuencia de valores que toman A, B y C.
ii. Mostrar los registros que se graban en el log en orden.
iii. Mostrar las actualizaciones que se hacen en memoria cache, asumiendo que la página que contiene a A y a B es regrabada al disco después del commit de T_3.
iv. Aplicar el algoritmo de recuperación *Deshacer/Rehacer* mostrando las listas que correspondan y los valores de cache y disco referidos a A, B y C.

NOTAS BIBLIOGRÁFICAS

Artículos importantes sobre control del paralelismo, el protocolo de dos fases y otros conceptos claves son los de K. P. Eswaran y otros [82]; C. H. Papadimitriou [170]; M. Yannakakis [214] y el más reciente de A. Thomasian [199].

Sobre recuperación en general cabe mencionar el artículo de T. Haerder y A. Reuter [124] mientras que A. Reuter [178] analiza el desempeño de diversas técnicas. Otros artículos significativos son los de H. Aghili y D. G. Severance [3]; P. A. Bernstein, A. N. Goodman y V. Hadzilacos [37] y J. Gray y otros [114]. C. Mohan y otros [166] presenta un reciente algoritmo denominado ARIES.

Existen también varios excelentes textos entre los que se destacan P. A. Bernstein, A. N. Goodman y V. Hadzilacos [38]; el más reciente aunque menos formal P. A. Bernstein y E. Newcomer [39]; V. Kumar [145]; C. H. Papadimitriou [171] y finalmente J. Gray y A. Reuter [116], considerado la Biblia del procesamiento de transacciones.

APÉNDICE

EL ESTÁNDAR SQL:1999

En el Capítulo 4 presentamos el lenguaje SQL que, como se dijo, se ajusta al estándar SQL92. A mediados de 1999 estaba a consideración de las organizaciones de estandarización, la estadounidense ANSI (American National Standards Institute) y la internacional ISO (International Standards Organization), la aprobación de una nueva versión del estándar para SQL conocido inicialmente como SQL3, y últimamente como SQL:1999. SQL3 comenzó a elaborarse en 1992 e incluye un gran número de nuevas características, algunas de las cuales existían ya, en productos comerciales, al momento de publicarse el estándar SQL92. Entre estas características encontramos las de orientación a objetos, tipos de datos multimedia, módulos persistentes almacenados, una "call-level interface" estándar, acceso remoto a los datos, facilidades de un lenguaje computacionalmente completo y facilidades para procesamiento transaccional, entre otras.

A.1. INTRODUCCIÓN

El lenguaje SQL:1999 es una extensión ascendentemente compatible con SQL92 con importantes cambios al modelo de datos. Sin embargo, los componentes fundamentales de los estándares anteriores no han cambiado. La relación o tabla sigue siendo el único medio para hacer persistente a los datos. Sólo que ahora una relación no requiere estar en 1°FN. Es decir, los componentes de las tuplas pueden ser grupos repetitivos y tipos de datos no atómicos, incluyendo filas, objetos y tipos de datos abstractos.

A efectos de clarificar la presentación de las novedades de SQL:1999, con respecto a SQL92, seguiremos el enfoque de Eisenberg y Melton [77], dividiéndolas en dos grandes grupos:

i. características relacionadas con la orientación a objetos y
ii. otras las que, a grandes rasgos, tienen que ver con el modelo relacional y con las características tradicionales del SQL.

Comenzaremos analizando las extensiones relacionales al SQL, indicadas en segundo término.

A.2. Características Relacionales

Estas características han sido organizadas en cinco grandes grupos, a saber, nuevos tipos de datos, nuevos predicados, semántica mejorada seguridad adicional y base de datos activa. A continuación analizaremos cada uno de esos grupos en detalle.

A.2.1. Nuevos Tipos de Datos

En SQL:1999 encontramos cuatro nuevos tipos de datos, con los cuales se extiende el sistema de tipos básico de SQL92. Estos nuevos tipos son:

LOB

El primer nuevo tipo de dato a analizar es Large OBject, o LOB, que tiene dos variantes: Character Large OBject, o CLOB y Binary Large OBject, o BLOB. Un BLOB es un portador, en la base de datos, de una pieza de datos binaria tal como una representación digitalizada de una fotografía o la firma de una persona. De la misma forma, un CLOB permite almacenar datos no estructurados en la bd, tales como un currículum vitae o una colección de artículos escritos por una persona.

Ambos tipos de datos proveen los mecanismos básicos para el almacenamiento y recuperación de datos multimedia. Por ejemplo, podríamos ampliar la tabla Médico para que incluya el cv del médico, su fotografía y la representación digital de su firma:


```
Create table Medico (
    hosp      smallint,
    numero    integer,
    nombre    varchar (30)
    cv        clob (50k),
    firma     blob (1M),
    foto      blob (10M)
);
```

Los tres últimos atributos son almacenados en la bd como LOB's.

Los LOB's tienen ciertas restricciones, ya que no admiten predicados como primary key o foreign key, ni de comparación que no sean la igualdad o desigualdad estricta. Tampoco pueden ser usados en las cláusulas group by u order by ni en la operación union. En cambio sí soportan las operaciones de almacenamiento y recuperación, y sus contenidos pueden ser recuperados parcialmente. LOB's con características similares pueden ser recuperados usando el predicado like. También los LOB's pueden ser concatenados y, en algunos casos, participar en la operación **union all**. En particular, a la variante CLOB es posible aplicarle las funciones de substring, posición y longitud.

Las aplicaciones pueden manipular valores de tipo LOB por medio de un tipo especial denominado LOB *locator*. Éstos son usados dentro de una aplicación para definir y ubicar unívocamente LOB's dentro de la bd. También pueden ser utilizados en operaciones, evitando así el movimiento innecesario de datos.

Boolean

El tipo de dato Boolean permite registrar directamente los valores de verdad *verdadero* (true), *falso* (false) y *desconocido* (unknown). Permite también la definición de funciones booleanas a usar en operadores y predicados definidos por el usuario. Como ejemplo, podríamos ampliar nuevamente la tabla Médico con un atributo que identifique si el médico trabaja en el hospital con tiempo completo:

```
Tiempo_completo    boolean
```

Luego podríamos listar los nombres de los médicos que trabajan tiempo completo en un hospital de la siguiente manera:

```
select nombre from Medico where tiempo_completo = TRUE;
```

Row

Este otro tipo de dato se define de manera similar a la definición de una tabla y, de hecho, es una extensión de ese concepto. El tipo row permite definir explícitamente un tipo estructurado dentro de la estructura de una tabla, con lo que se violaría la 1°FN. Sin embargo, según Eisenberg y Melton, la mayoría de los observadores opina que esto no es más que un tipo de dato que admite su descomposición, tal como una cadena de caracteres.

Un tipo row se define por su nombre y una secuencia de pares < nombre atributo, tipo de dato>. Si deseáramos agregar, por ejemplo, la dirección detallada del hospital en la tabla Hospital podríamos escribir:

```
create row type   direccion_def (
    calle         varchar (30)
    nro           int,
    ciudad        varchar (30),
    provincia     varchar (30),
    cod_postal    int
    );
```

Luego, en la definición de la tabla Hospital escribimos:

```
create table Hospital (
    numero      smallint not null
    nombre      char (30),
    direccion   direccion_def,
    telef       int,
    camas       smallint
    );
```


Una manera alternativa de definir lo mismo es la siguiente:

```
create table Hospital (
    numero      smallint not null
    nombre      char (30),
    direccion row (
                calle          varchar (30),
                nro            int,
                ciudad         varchar (30),
                provincia      varchar (30),
                cod_postal     int
                ),
    telef       int,
    camas       smallint
    );
```

Si ahora quisiéramos listar las ciudades a las que corresponden los hospitales escribimos

```
select direccion.ciudad from Hospital;
```

Array

El tipo de dato array se especifica como un tipo colección, proporcionando así un medio para especificar una lista de elementos para un atributo. Los elementos de un array pueden ser de cualquiera de los tipos existentes, por ejemplo integer, row, etc. Si en la definición de la tabla Hospital necesitáramos registrar hasta cinco números de teléfono, podríamos escribir:

```
create table Hospital (
    numero      smallint not null
    nombre      char (30),
    direccion   direccion_def,
    telef       int   array [5],
    camas       smallint
    );
```

También en este caso podríamos decir que este tipo de dato viola la 1°FN, al permitir grupos repetitivos. Al igual que con row vale el argumento que array es un tipo que admite ser descompuesto

A.2.2. Nuevos Predicados

Tres son los nuevos predicados de SQL:1999. En este punto veremos sólo dos, mientras que el tercero será analizado cuando tratemos las extensiones relacionadas con la orientación a objetos.

El primero de los predicados se denomina similar y permite usar expresiones regulares en operaciones de búsqueda en textos. Con este predicado se puede especificar una cadena de caracteres y buscar por algo similar a esa cadena, dentro de algún grado de similitud. Un ejemplo, tomado de la obra citada de Eisenberg y Melton, es el siguiente:

```
where nombre similar to '(SQL-(86 | 89 | 92 | 99)) | (SQL(1 | 2 | 3))
```

donde este predicado se satisfaría para cualquiera de los nombres que recibió el estándar SQL a través de los años.

El segundo predicado es distinct y opera de manera similar a unique salvo que cuando distinct se aplica a dos valores nulos el resultado es falso.

A.2.3. Nueva Semántica

Existen varios nuevos aspectos del comportamiento de SQL:1999, de los cuales analizaremos sólo dos: recursión y *savepoints*. Otros son extensiones a la actualización de vistas y al procesamiento de transacciones.

SQL Recursivo

Esta extensión permite escribir en SQL expresiones que involucran tablas vistas autorreferenciadas, con lo que se facilitan las consultas referidas a problemas tales como el recorrido de redes y las listas de materiales. Recordemos el ejercicio 8 del Capítulo 4 en el que planteamos un esquema BOM-

(componente, subcomponente, cantidad). Por medio de la tabla BOM y utilizando el concepto de consulta recursiva, podemos ahora fácilmente encontrar todos los elementos que componen un determinado producto, por ejemplo el que tiene código LP3D042.

```
with recursive lista_m (componente, subcomponente) as
    (select componente, subcomponente
        from   BOM
    union
    select l.componente, b.subcomponente
        from lista_m l, BOM b
            where l.subcomponente = b.componente)
select * from lista_m where componente = "LP3D042";
```

Observemos que la expresión se inicia con la palabra clave with seguido por la palabra clave recursive. La expresión continúa con la definición de la tabla recursiva: el nombre de la relación, la palabra clave as y la consulta que define la relación. Finalmente encontramos una consulta que se refiere a la tabla antes definida y cuya evaluación produce el resultado de toda la expresión.

Este nuevo elemento del lenguaje no está exento, sin embargo, de ciertos problemas, por lo que su uso debe realizarse con cierto cuidado, particularmente ante la presencia de negación, agrupamiento y agregación, los operadores intersect y except, etc. Otro de los problemas es el derivado del tamaño de los conjuntos que deberán ser analizados, ya que si se encuentra un ciclo, el tamaño de los conjuntos puede resultar ilimitado. La cláusula cycle permite manejar este problema al posibilitar restringir la profundidad de una búsqueda recursiva.

Savepoints

Si bien el concepto de *savepoint* es nuevo en el estándar SQL:1999, ya había sido implementado en diversos productos comerciales al publicarse el anterior estándar. Un savepoint es un punto dentro de una transacción al cual es posible retornar, ya que realiza una marca intermedia del estado de esa transacción. Un savepoint permite particionar la ejecución de una transacción de larga duración en puntos manejables en el tiempo. Este concep-

to es aplicable cuando la transacción involucra varias funciones y se aplicaría un savepoint antes de realizar cada función. Luego, si en algún punto una función falla, la transacción simplemente retorna al estado previo al comienzo de esa función.

El formato de la sentencia savepoint es

savepoint <nombre del savepoint>.

Cuando se detecta la sentencia savepoint dentro de una transacción, ésta se ejecuta grabándose en el log del sistema, en memoria no volátil, los registros del log que se encontraban en el buffer más un registro de savepoint; pero esto limitado a los registros de la transacción que ejecutó el savepoint.

La razón para crear un savepoint es retrotraer las acciones de una transacción a un punto previo. Para esto se cuenta con la sentencia rollback to savepoint <nombre del savepoint>.

También existe una sentencia release savepoint <nombre del savepoint>, cuyo efecto es eliminar la marca del savepoint en el log.

A.2.4. Seguridad

Con el fin de mejorar los aspectos relacionados con la seguridad se ha introducido la noción de *rol*. Se define un rol como una colección nominada de privilegios relacionados, los que pueden ser otorgados a usuarios u otros roles. Los roles han sido diseñados para facilitar la administración de los privilegios sobre los diversos objetos.

Los roles pueden ser creados para acceder a las aplicaciones o para administrar los privilegios de un grupo. Los roles de aplicación dan derechos aprobados a un grupo de usuarios para ejecutar una aplicación determinada. Las aplicaciones pueden tener varios roles y cada uno puede tener un conjunto de privilegios diferentes. Por otra parte, los roles de usuario se crean para agrupar privilegios comunes a un grupo de usuarios.

Los roles tienen las siguientes propiedades:

i. Pueden ser otorgado a otros roles; pero no a sí mismos, directa o indirectamente.

ii. Pueden ser otorgados a uno o más usuarios.

iii. Pueden ser habilitados o inhabilitados

iv. Pueden ser revocados desde uno o más usuarios o roles.

Para crear un rol se cuenta con una sentencia cuyo formato es:

create rol <nombre de rol>

Para esto se debe contar con el privilegio necesario otorgado por el administrador de la bd. Quien crea un rol es, habitualmente, el mismo administrador o el creador de un objeto de la bd.

La sentencia grant se usa para otorgar acceso al sistema u objetos de un esquema a usuarios o roles dentro del sistema. En el siguiente ejemplo

```
create role usuario;
grant select on Hospital to usuario;
```

la primer sentencia crea el rol usuario, mientras la segunda otorga la operación select al rol usuario.

Una vez otorgado un rol o privilegio a un usuario o rol, es posible revocar el privilegio, por medio del operador revoke. Por ejemplo, si se desea revocar la habilidad de los usuarios para hacer select de la tabla Hospital, escribimos:

```
revoke select on Hospital from usuario;
```

Si por otra parte, se desea revocar todos los derechos otorgados al rol usuario desde el usuario llamado Hugo, escribimos:

```
revoke usuario from Hugo;
```

Para eliminar un rol de la bd se cuenta con el operador drop. Si, por ejemplo, se desea eliminar el rol usuario, usamos la sentencia

```
drop role usuario;
```

A.2.5. Base de Datos Activa

Aunque la noción de base de datos activa ha existido por años en los productos comerciales, recién ha sido reconocida en esta versión del estándar. De esta manera, SQL:1999 establece un cierto orden en este aspecto, ya que los productos tenían entre sí diferencias semánticas, en muchos casos significativas.

Las bases de datos activas ofrecen todas las funcionalidades de una base de datos tradicional y, además, son capaces de reaccionar, con o sin la intervención del usuario, a las modificaciones del estado de la bd o de su entorno. Esta capacidad de reacción es debida a la integración de reglas de producción dentro de la bd. Esas reglas son desencadenadas por las transiciones de estado dentro de la bd, inmediatamente después que ellas evalúan los predicados que son definidos sobre el estado de la bd, y ejecutan las acciones asociadas. Las reglas de producción permiten la definición de operaciones que son ejecutadas automáticamente cuando se producen ciertos eventos o cuando se satisfacen ciertas condiciones. Las reglas de producción son implementadas en SQL:1999 por medio de objetos de la bd denominados triggers. Luego un trigger es un objeto nominado de la bd, una pieza de software que se activa en base a un evento especificado. Un trigger tiene las siguientes características:

i. Se activa solamente cuando ocurre un cierto evento. Los tipos de eventos soportados por SQL:1999 son inserción, eliminación y actualización de tuplas, columnas o tablas.
ii. Verifica la condición para la que fue diseñado.
iii. Si la condición es válida se ejecutan las acciones definidas en el cuerpo del trigger. De lo contrario no se realiza ninguna acción.
iv. Se puede especificar que ocurra antes o después que el evento para el que fue definido. Además, se puede establecer que ocurra en lugar del evento que lo invocó.
v. Puede causar la invocación de otro trigger.

La forma general de un trigger incluye el nombre del trigger, el evento que lo invoca, una especificación de granularidad, una condición para que se ejecute y una acción:


```
Create trigger <nombre del trigger>                          nombre
    before | after <evento> on <nombre de tabla>             evento
    referencing new as <nombre de una variable tupla>        granularidad
                old as <nombre de una variable tupla>
    for each [row | statement]
    when <condición>                                         condición
      begin                                                  acción
      ...
      end;
```

Veamos un ejemplo: supongamos que se admite la modificación de un código de hospital en la tabla Hospital. Luego, ante una modificación tal deberíamos inmediatamente modificar el código de hospital en la tabla Médico, para todos los médicos que trabajan en ese hospital. La primer modificación, en la tabla Hospital, la confiamos a una persona; pero las de la tabla médico podemos realizarlas automáticamente. Para ello definimos un trigger de la siguiente forma:

```
Create trigger mod_hosp
        After update of codigo on hospital
        Referencing old as ho,
                    new as hn
        for each row
            update Medico
            set hosp = hn.codigo
            where hosp = ho.codigo;
```

Esta definición crea un trigger con nombre "mod_hosp". Este trigger será invocado después de la modificación del atributo "codigo" en la tabla Hospital. La tupla antigua (antes de la modificación) la denominamos "ho" y con "hn" indicamos la nueva tupla (después de la modificación). Luego se indica que para cada tupla actualizada se ejecutará la actualización de la tabla Médico. Esta última se realizará, a su vez, con cada tupla tal que su código de hospital, en este caso "hosp", cuyo valor coincida con el valor

antiguo de "codigo" (ho.codigo). Luego se ejecuta el set colocando en la columna 'hosp" el nuevo número de hospital que se encuentra en hn.codigo.

A.3. Características Relacionadas con Orientación a Objetos

En esta sección analizaremos las extensiones realizadas al estándar a efectos que SQL soporte los conceptos de orientación a objetos. Entre las nuevas características se cuentan los tipos estructurados definidos por el usuario y los módulos almacenados persistentes.

A.3.1. Tipos Estructurados Definidos por el Usuario

Después de la ampliación del concepto de tabla, la extensión más significativa de SQL:1999 es el *tipo de dato abstracto* (ADT). La especificación de un ADT involucra la especificación de los atributos y la especificación de las funciones: ambas están contenidas en la definición de un ADT. Tanto la estructura como el comportamiento de un ADT están completamente encapsulados y lo único visible externamente son las definiciones de los atributos y las interfaces de las funciones.

Un ADT puede ser especificado como valor o como objeto. Un ADT valor no posee una identidad, sino que es sólo un valor. Un ADT objeto se caracteriza porque cada instancia tiene un identificador de objeto u OID que lo identifica unívocamente. Por otra parte, los ADT's pueden ser usados como tipos de datos en la definición de atributos de una relación, como atributos de otro ADT, como parámetros en *procedures* o funciones, y como variables en sentencias de SQL.

Veamos un ejemplo de la definición de atributos de ADT:

```
Create type direccion_d (
        calle        varchar (30),
        nro          int,
        ciudad       varchar (30),
        provincia    varchar (30),
        cod_postal   int)
```



```
create type hospital (
        numero       smallint not null
        nombre       char (30),
        direccion    direccion_d,
        telef        int,
        camas        smallint)
```

El acceso e inicialización de los atributos del ADT definido, son realizados por medio de funciones generadas automáticamente. Esas funciones se denominan *observadores* y *mutadores*. En el ejemplo siguiente se muestran las interfaces para las funciones de algunos de los atributos del ADT *direccion*:

```
Function calle (direccion_d) returns varchar (30)
Function calle (direccion_d, varchar (30) ) returns direccion

Function ciudad (direccion_d) returns varchar (30)
Function ciudad (direccion_d, varchar (30)) returns direccion

Function cod_postal (direccion_d) returns int
Function cod_postal (direccion_d, int) returns direccion
```

Esas funciones pueden ser usadas para acceder o modificar valores de atributos, como se muestra a continuación:

```
c = cod_postal (direccion_d)      / la función observadora de cod_postal
                                  devuelve en c el código postal, por ej. 1416 /
p = cod_postal (direccion_d, 1405)  / devuelve una instancia completa de
                                  dirección con el nuevo código postal /
```

Los atributos de un ADT se definen mediante la especificación de un nombre y un tipo. El tipo de un atributo puede ser cualquiera de los tipos predefinidos de SQL, como así también otros tipos definidos por el usuario, tal como row, u otro ADT. Éste es el caso del atributo *direccion* en la tabla Hospital, cuyo tipo es otro ADT, *direccion_d*.

Además de los atributos, un ADT puede tener otros varios componentes. Un ADT se puede definir como un subtipo de otro ADT. Con este fin se espe-

cifica en la definición del primero la cláusula under y el nombre del otro ADT que cumple la función de supertipo

Las funciones *constructor* y *destructor*, asociadas a un ADT, se usan para crear y eliminar una instancia del ADT respectivamente, en la base de datos. Estas funciones son creadas automáticamente. Sin embargo, es posible definir otras funciones con el mismo fin, independientemente de aquéllas. En el ejemplo siguiente mostramos las funciones de construcción y destrucción de *direccion_d*:

```
constructor function direccion_d
        ( a varchar (30), n int, c varchar (30),
        p varchar (30), l int)) return direccion_d
        declare :d direccion_d
        begin
                new :d;
                set :d calle = a;
                set :d nro  = n;
                set :d ciudad = c;
                set :d provincia = p;
                set :d cod_postal = l;
                return :d;
        end;
end function

destructor function elim_direccion ( :d direccion_d)
                        returns direccion_d
                begin
                destroy :d;
                return :a;
                end;
        end function
```

Como vimos, las funciones de observación y mutación permiten acceder a los valores de atributos de un ADT. Éstas pueden ser usadas de dos maneras diferentes. Una primera forma de uso ha sido ejemplificada más arriba. Otro método es usar la notación *punto*, en la cual usamos la variable de instancia del ADT seguida por un punto y luego el nombre de la función que se invoca. Por ejemplo, para acceder al atributo calle de la dirección de un hospital escribimos:


```
Set cl = h direccion.calle;
```

donde la variable h ha sido declarada de tipo hospital y la variable cl almacenará el valor del atributo.

La interfaz del ADT

Los atributos y rutinas de un ADT pueden ser declarados con un determinado grado de visibilidad. Esas declaraciones restringen la forma de uso de los datos, permitiendo a los diseñadores controlar cómo los usuarios externos utilizan los atributos y funciones encapsulados. En SQL:1999 se han definido tres posibles grados de visibilidad: *público, privado* y *protegido*.

El primer nivel de visibilidad es *público*. Los atributos y funciones declarados públicos, si bien no son directamente accesibles, lo son a través de las funciones de observación y mutación. El atributo o función debe ser precedido por la declaración public.

El segundo nivel de visibilidad se obtiene por medio de la declaración de privado usando la palabra clave private. Un atributo o función se considera *privado* si sólo se puede utilizar dentro del ADT donde ha sido declarado.

El último nivel es *protegido*. En la declaración se usa la palabra clave protected. Un atributo o función es *protegido* de ser visto o modificado por usuarios externos. Un atributo o función protegido es accesible, sin embargo, desde cualquier tabla, rutina o ADT que usa los atributos como un subtipo.

Subtipos y Herencia

Ya mencionamos el uso de la cláusula under en la definición de un ADT. El tipo usado como supertipo es también un ADT, por lo que podría haber sido definido usando otro ADT, y así siguiendo. Los ADT definidos de esta forma heredan la estructura (atributos, etc.) y el comportamiento (funciones o métodos) de los supertipos. SQL:1999 ha adoptado el modelo de herencia

simple, es decir, la jerarquía de herencia es un árbol. Luego cada ADT hereda de un supertipo único. El tipo que hereda puede tener sus propios atributos como así también sus propias funciones. Las funciones o métodos definidos en el nivel del subtipo pueden sobrecargar las definiciones de funciones en el nivel del supertipo

Funciones o Métodos

Además de las funciones que se generan automáticamente, un ADT puede contar con otras funciones, las que se definen en el cuerpo del ADT. Estas funciones permiten proporcionar la funcionalidad que los usuarios de un ADT pueden necesitar. Los usuarios son los desarrolladores de aplicaciones que tienen acceso a la signatura del ADT. El resultado deseado se obtiene invocando la función con los parámetros adecuados.

Los nombres de las funciones pueden ser sobrecargados. O sea que pueden existir varias instancias diferentes para un mismo nombre de función. Un ejemplo es la función *display*, la que puede exhibir distintas cosas, desde un BLOB o un CBLOB, hasta un entero.

En el caso que varias funciones sean implementadas con el mismo nombre, en un ADT, la ejecución de la función apropiada se basa en los tipos corrientes de los argumentos pasados en la invocación

Las funciones pueden proporcionar el comportamiento de un ADT desde la interfaz del mismo o pueden ser código separado, o parte de un procedimiento o módulo SQL. Pueden usar ADT's o código embebido para realizar alguna operación determinada sobre la bd o sobre algún ADT o estructura de datos dada. La ventaja de los procedimientos o funciones embebidas en un ADT es tener acceso a las funciones y procedimientos del ADT que pueden no ser visibles en la interfaz del ADT.

A.3.2. Módulos Persistentes Almacenados

La facilidad de módulos persistentes almacenados (PSM) de SQL posee construcciones sintácticas y semánticas que permiten la especificación de módulos almacenados, procedimientos y funciones. Los módulos PSM son computacionalmente completos, por lo que no se requiere un lenguaje externo para el desarrollo de una aplicación. Incluye extensiones procedimentales de los siguientes tipos: declaración de variables; sentencias compuestas; sentencias de control de flujo como asignación, if, case; la sentencia iterativa for y sentencias para control de ciclos como loop, while y repeat, entre otras.

La primer versión de SQL/PSM fue liberada hace ya unos años como una extensión al SQL92. Esta nueva versión presenta una importante mejora con la introducción de métodos, los que son un tipo especial de función. Las siguientes son algunas características de los métodos, en contraposición a las funciones o rutinas comunes: están ligados a un único ADT; cuentan con un argumento no declarado (el primero) que tiene el tipo de dato del ADT al que se encuentra ligado; se almacenan en el mismo esquema que su correspondiente ADT; pueden ser polimórficos, en cuyo caso la determinación exacta del método invocado puede diferirse hasta el momento de su ejecución.

Podemos definir a los módulos persistentes de SQL como colecciones encapsuladas de rutinas y declaraciones. Entre estas últimas podemos incluir variables, cursores, tipos, tablas, vistas, funciones y procedimientos. Siendo que esas colecciones son escritas en un SQL computacionalmente completo el problema conocido como "desfasaje de impedancia" prácticamente ha dejado de existir.

Las rutinas en SQL:1999 son componentes de los módulos del servidor de la bd. En consecuencia, son objetos del esquema y por lo tanto son almacenadas en forma persistente en el ámbito de la bd. Tanto su definición como su estado están disponibles desde la bd y no es necesario interacción exterior para ser accedidas.

Extensiones Procedimentales

A continuación analizaremos brevemente las principales extensiones procedimentales incorporadas al SQL:1999.

Sentencias Compuestas

La estructura de sentencia compuesta permite agrupar sentencias SQL en una unidad por medio de las palabras claves begin y end. Las sentencias agrupadas se ejecutan secuencialmente y pueden ser consideradas como una unidad atómica o como sentencias aisladas. En el primer caso, la falla de cualquier sentencia en la sentencia compuesta produce la falla de todo el gru-

po. En el segundo caso, la falla de una sentencia no provoca la falla de las restantes. Si se desea que la ejecución sea atómica, la palabra atomic debe ser incluida a continuación de begin. De esta manera es posible elegir entre comportamiento atómico y no atómico.

Esta estructura permite ejecutar varias sentencias SQL con una única sentencia exec SQL en un programa huésped.

Para definir una variable dentro de una sentencia compuesta, se utiliza la palabra clave declare seguida por el nombre de la variable y el tipo asociado. declare también se puede utilizar en funciones y procedimientos. Las variables que han sido declaradas en una sentencia compuesta son creadas e inicializadas apropiadamente al comenzar su ejecución. Al finalizar la ejecución, las variables son destruidas.

Dentro de una sentencia compuesta también es posible declarar cursores los que, al igual que las variables, son creados y destruidos al comenzar y al finalizar la ejecución de la sentencia, respectivamente.

Finalmente, una sentencia compuesta puede especificar un conjunto de "manipuladores de condición". En general, un manipulador especifica las condiciones que puede manejar, la acción que tomará si se cumple alguna de las condiciones y el efecto que tiene el manipulador, tal como continue, exit o undo, si la acción se ejecuta con éxito.

Sentencias de Flujo de Control

La sentencia de asignación permite asignar el resultado de una expresión a una variable. Por ejemplo:

```
declare nombre_h char (30);
set nombre_h = (select nombre from Hospital
                where numero = 15);
```

que asigna a *nombre_h* el nombre del hospital número 15.

If – then – else es una forma de ejecución condicional y constituye una de las más importantes sentencias incorporadas a SQL:1999. Su forma básica es la siguiente:


```
If condición verdadera
    then sentencia SQL
    elseif condición verdadera
    then sentencia SQL
    ...
    elseif condición verdadera
    then sentencia SQL
    else sentencia SQL
end if
```

La sentencia case presenta dos variantes. En la primera simplemente se evalúa una expresión y en base el resultado de la evaluación, se sigue uno de los caminos definido en la lista de alternativas. En la segunda variante, cada alternativa tiene su propia condición de búsqueda y actúa de manera similar a if – then – else.

La sentencia while proporciona la ejecución repetida de un grupo de sentencias, siempre que se verifique una determinada condición. La prueba de que la condición se verifica es realizada al comienzo de la sentencia.

La sentencia for, a diferencia de las anteriores, es bastante diferente a la hallada en la mayoría de los lenguajes de programación. For itera sobre los elementos de una tabla resultado, para todas las filas de la tabla. La finalidad es ejecutar un grupo de sentencias SQL para cada fila de la tabla. De esta manera, es similar a una definición implícita de un cursor, con fetches sobre todas las filas.

Finalmente mencionamos las sentencias de control de ciclos loop y repeat. La sentencia loop permite construir una estructura de control para la ejecución de un grupo de sentencias en forma continua. La única manera de salir del ciclo es por medio de la sentencia leave. Distinto de la anterior, la sentencia repeat requiere una condición de terminación. De esta forma, las sentencias dentro del ciclo se repiten hasta que se cumple la condición especificada al final de la secuencia; por lo que se garantiza que la secuencia completa de sentencias, dentro del cuerpo del repeat, se ejecutará al menos una vez.

A.4. IMPLEMENTACIONES

Existen diversos productos tales como ORACLE 8 Universal Data Server, de Oracle Corporation, DB2 Version 5 Universal Database, de IBM e Informix Universal Server, de Informix Software, que implementan el modelo Objeto-Relacional y que, en diversa medida, soportan el estándar SQL:1999. Se espera, sin embargo, que los diversos fabricantes de sgbd adhieran al estándar, al menos, a un conjunto básico de características que se conoce como CORE SQL. Los productos que soportan CORE SQL, aunque no todo el estándar, se denominan *entry-level compliant*.

NOTAS BIBLIOGRÁFICAS

El núcleo de este apéndice sigue a Eisenberg y Melton [77], miembros del comité que desarrolló el estándar SQL:1999. Para más detalles es posible recurrir a las publicaciones de ISO y ANSI. Los principales componentes de la publicación son ISO/IEC 9075:1999 Framework, Foundation, Call-Level Interface, Persistent Stored Modules y Host Language Bindings.

BIBLIOGRAFÍA

[1] Abiteboul, S. y R. Hull: "IFO: a formal semantic data model", *Third ACM Symp. on Principles of Database Systems*, pág. 119-132. 1987.

[2] Abiteboul, S., R. Hull, y V. Vianu: *Foundations of Databases*. Addison-Wesley. 1995.

[3] Aghili, H. y D. G. Severance: "A practical guide to the differential files for recovery of on-line databases", *ACM Trans. on Database Systems 7:4*, pág. 540-565. 1982.

[4] Agrawal, R., M. J. Carey, y M. Linvy: "Models for studying concurrency control performance: alternatives and implications", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 108-121. 1985.

[5] Agrawal, R. y D. J. DeWitt: "Integrated concurrency control and recovery mechanisms: design and performance evaluation", *ACM Database Systems* 10:4, pág. 529-564. 1985.

[6] Aho, A. V., C. Beeri, y J. D. Ullman: "The theory of joins in relational databases", *ACM Trans. on Database Systems* 4:3, pág. 297-314. 1979.

[7] Aho, A. V., J. E. Hopcroft, y J. D. Ullman: *The Design and Analysis of Computer Algorithms*. Addison-Wesley. 1974.

[8] Alagic, S.: *Relational Database Technology*. Springer-Verlag. 1986.

[9] Armstrong, W. W.: "Dependency structures of database relationship", *Proc. IFIP Congress*, pág. 580-583. 1974.

[10] Astraham, M. M. y D. D. Chamberlin: "Implementation of a structured English query language", *Comm. ACM* 18:10, pág. 580-587. 1975.

[11] Astraham, M. M., et al.: "System R: a relational approach to data management", *ACM Trans. on Database Systems* 1:2, pág. 97-137. 1976.

[12] Astraham, M. M., et al.: "System R: a relational database management system", *Computer* 12:5, pág. 43-48. 1979.

[13] Atzeni, P. y V. De Antonellis: *Relational Database Theory*. Benjamin-Cummings. 1993.

[14] Bachman, C. W.: "Data structure diagrams", *Data Base* 1:2, pág. 4-10. 1969.

[15] Bancilhon, F. y S. Khoshafian: "A calculus for complex objects", Proc. *ACM Symp. on Principles of Database Systems*, pág. 53-59. 1986.

[16] Bancilhon, F. y R. Ramakrishnan: "An amateur's introduction to recursive query-processing strategies", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 16-52. 1986.

[17] Batini, C., S. Ceri, y S. Navathe: *Conceptual Database Design: An Entity-Relationship Approach*. Addison-Wesley. 1992.

[18] Bayer, P. y E. M. McCreight: "Organization and maintenance of large ordered indices", *Acta Informatica* 1:3, pág. 173-189. 1972.

[19] Bayer, P. y M. Schkolnick: "Concurrency of operating on B-trees", *Acta Informatica* 9:1, pág. 1-21. 1977.

[20] Beeri, C.: "On the membership problem for functional and multivalued dependencies", *ACM Trans. on Database Systems* 5:3, pág. 241-259. 1980.

[21] Beeri, C. y P. A. Bernstein: "Computational problems related to the design of normal form relation schemes", *ACM Trans. on Database Systems* 4:1, pág. 30-59. 1979.

[22] Beeri, C., P. A. Bernstein, y N. Goodman: "A sophisticate's introduction to database normalization theory", *Proc. Intl. Conf. on Very Large Data Bases*, pág. 113-124. 1978.

[23] Beeri, C., P. A. Bernstein, N. Goodman, M. Y. Lai, y D. E. Shasha: "A concurrency control theory for nested transactions", *Proc. Second ACM Symp. on Principles of Database Systems*, pág. 45-62. 1983.

[24] Beeri, C., P. Fagin, y J. H. Howard: "A complete axiomatization for functional and multivalued dependencies", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 47-61. 1977.

[25] Beeri, C. y P. Honeyman: "Preserving functional dependencies", *SIAM J. Computing* 10:3, pág. 647-656. 1981.

[26] Beeri, C., A. O. Mendelzon, Y. Sagiv, y J. D. Ullman: "Equivalence of relational database schemes", *SIAM J. Computing* 10:2, pág. 352-370. 1981.

[27] Beeri, C., S. Naqvi, R. Ramakrishnan, O. Shmueli, y S. Tsur: "Sets and negation in a logic database language (LDL1)", *Proc. Sixth ACM Symp. of Principles of Database Systems*, pág. 21-37. 1987.

[28] Beeri, C. y M. Y. Vardi: "The implication problem for data dependencies", *Automata, Languages y Programming* (S. Even y O. Kariv, eds.), pág. 73-85. Springer-Verlag, New York. 1981.

[29] Beeri, C. y M. Y. Vardi: "Formal systems for tuple- and equality-generating dependencies", *SIAM J. Computing* 13:1, pág. 76-98. 1984.

[30] Beeri, C. y M. Y. Vardi: "A proof procedure for data dependencies", *J. ACM* 31:4, pág. 718-741. 1984.

[31] Bentley, J. L.: "Multidimensional binary search trees used for associative searching", *Comm. ACM* 18:9, pág. 507-517. 1975.

[32] Bentley, J. L. y J. H. Friedman: "Data structures for range searching", *ACM computing Surveys* 11:4, pág. 397-410. 1979.

[33] Bernstein, P. A.: "Synthesizing third normal form relations from functional dependencies", *ACM Trans. on Database Systems* 1:4, pág. 277-298. 1976.

[34] Bernstein, P. A. y N. Goodman: "What does Boyce-Codd normal form Do?", *Proc. Intl. Conf. on Very Large Data Bases*, pág. 245-259. 1980.

[35] Bernstein, P. A. y N. Goodman: "Concurrency control in distributed systems", *Computing Surveys* 13:2, pág. 185-221. 1981.

[36] Bernstein, P. A. y N. Goodman: "Multiversion concurrency control: theory and algorithms", *ACM Trans. on Database Systems* 8:4, pág. 463-483. 1983.

[37] Bernstein, P. A., N. Goodman, y V. Hadzilacos: "Recovery algorithms for database systems", *Proc. 1983 IFIP Congress*, pág. 799-807. North Holland, Amsterdam. 1983.

[38] Bernstein, P. A., V. Hadzilacos, y N. Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. 1987.

[39] Bernstein, P. A. y E. Newcomer: *Principles of Transaction Processing*. Morgan Kaufmann. 1997.

[40] Biliris, A.: "Operation specific locking in B-trees", *Proc. Sixth ACM Symp. on Principles of Database Systems*, pág. 159-169. 1987.

[41] Biskup, J., U. Dayal, y P. A. Bernstein: "Synthesizing independent da-

tabase schemas", *ACM SIGMOD Intl. Conf. on Management of* Data. pág. 143-152. 1979.

[42] Bocca, J.: "EDUCE: a marriage of convenience: Prolog and a Relational Database", *Symp. on Logic Programming*. pág. 36-45. IEEE. New York. 1986.

[43] Bolour, A.: "Optimality properties of multiple key hashing functions", *J. ACM* 26:2, pág. 196-210. 1979.

[44] Boyce, R. F., D. D. Chamberlin, W. F. King, y M. M. Hammer: "Specifying queries in relational expressions: the SQUARE data sublanguage", *Comm. ACM* 18:11, pág. 621-628. 1975.

[45] Brachman, R. J. y H. J. Levesque: *Readings in Knowledge Representation*. Morgan Kaufmann. 1985.

[46] Brodie, M. L. y J. Mylopoulos: *On Knowledge Base Management Systems*. Springer-Verlag, New York. 1986.

[47] Brodie, M. L., J. Mylopoulos, y J. W. Schmidt: *On Conceptual Modeling*. Springer-Verlag, New York. 1984.

[48] Buckley, G. N. y A. Silberschatz: "Beyond two-phase locking", *J. ACM* 32:2, pág. 314-326. 1985.

[49] Cardenas, A. F.: *Data* Base *Management Systems*. Allyn y Bacon, Boston, Mass. 1985.

[50] Casanova, M. A., R. Fagin, y C. H. Papadimitriou: "Inclusion dependencies and their interaction with functional dependencies", *J. Computer y System Sciences* 28:1, pág. 29-59. 1984.

[51] Castano, S., M. Fugini, G. Martella, y P. Samarati: *Database Security*. Addison-Wesley. 1995.

[52] Ceri, S. y G. Pelagatti: *Distributed Databases. Principles and Systems*. McGraw-Hill, New York. 1984.

[53] Chamberlin, D. D., et al.: "SEQUEL 2: a unified approach to data definition, manipulation, and control", *IBM J. Research y Development* 20:6, pág. 560-575. 1976.

[54] Chamberlin, D. D., et al.: "A history and evaluation of System R", *Comm ACM* 24:10, pág. 632-646. 1981.

[55] Chandra, A. K. y D. Harel: "Computable queries for relational database systems", *J. Computer y System Sciences* 21:2 pág. 156-178. 1980.

[56] Chandra, Q. K. y D. Harel: "Structures and complexity of relational queries" *J. Computer y System Sciences* 25:1, pág. 99-128. 1982.

[57] Chandra, A. K., H. R. Lewis, y J. A. Makoswky: "Embedded implicational dependencies and their inference problem", *Proc. Thirteenth Annual ACM Symp. on the Theory of Computing*, pág. 342-354. 1981.

[58] Chandy, K. M., J. C. Browne, C. W. Dissly, y W. R. Ubrig [1975]: "Analytic models for rollback and recovery strategies in database systems", *IEEE Transactions on Software Engineering* SE-1:1, pág. 100-110. 1975.

[59] Chaudhuri, S. y K. Shim: "Optimization queries with aggregate views", *Proc. Intl. Conf. on Extending Database Technology*. 1996.

[60] Chen, P. P: "The entity-relationship model: toward a unified view of data", *ACM Trans. on Database Systems* 1:1, pág. 9-36. 1976.

[61] Clark, K. L.: "Negation as failure", en Gallaire y Minker, pág. 293-322. 1978.

[62] Codd, E. F.: "A relational model for large shared data banks", *Comm. ACM* 13:6, pág. 377-387. 1970.

[63] Codd, E. F: "Normalized Database structure: A brief tutorial, *ACM SIGFIDET Workshop on Data Description Access and Control*. 1971.

[64] Codd, E. F.: "Further normalization of the data base relational model", en *Data* Base *Systems* (R. Rustin, ed.), pág. 33-64. Prentice-Hall, Englewood Cliffs, New Jersey. 1972.

[65] Codd, E. F.: "Relational completeness of data base sublanguages", ibid., pág. 65-98.

[66] Codd, E. F. [1975]: "Understanding relations", *FDT* 7:3-4, pág. 23-28, ACM, New York. 1975.

[67] Codd, E. F.: "Extending the data base relational model to capture more meaning", *ACM Trans. on Database Systems* 4:4, pág. 397-434. 1979.

[68] Comer, D.: "The difficulty of optimum index selection", *ACM Trans. on Database Systems* 3:4, pág. 440-445. 1978.

[69] Comer, D.: "The ubiquitous B-tree", *Computing Surveys* 11:2, pág. 121-138. 1978.

[70] Culik, K., T. Ottmann, y D. Wood: "Dense multiway trees", *ACM Trans. on Database Systems* 6:3, pág. 486-512. 1981.

[71] Date, C.: *An Introduction to Database Systems* (6ª ed.), Addison Wesley. 1995.

[72] Dayal, U. y P. A. Bernstein: "On the correct translation of update ope-

rations on relational views", *ACM Trans. on Database Systems* 7:3, pág. 381-416. 1982.

[73] Delobel C.: "Normalization and hierarchical dependencies in the relational data model", *ACM Trans. on Database Systems* 3:3, pág. 201-222. 1978.

[74] Delobel, C. y R. C. Casey [1972]: "Decomposition of a database and the boolean switching functions", *IBM J. Res. Devel.* 17:5, pág. 370-386. 1972.

[75] DeWitt, D. J. et al.: "Implementation techniques for main memory database systems", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, pág. 1-8. 1984.

[76] Diederich, J. y J. Milton: "New methods and fast algorithms for database normalization", *ACM Trans. on Database Systems* 13:3, pág. 339-365. 1988.

[77] Eisenberg A.-Melton, J.: "SQL:1999, formerly known as SQL3", *SIGMOD Record* 28:1, pág. 131-138. 1999.

[78] Ellis, C. S.: "Concurrent search and insertion in 2-3 trees", *Acta Informatica* 14:1, pág. 63-86. 1980.

[79] Ellis, C. S.: "Concurrency in linear hashing", *ACM Trans. on Database Systems* 12:2, pág. 195-217. 1987.

[80] Elmasri R. y S. Navathe: *Fundamentals of Database Systems* (3ª ed.). Addison-Wesley. 1998.

[81] Elmasri R. y G. Wiederhold: "Data model integration using the structural model", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 191-202. 1979.

[82] Eswaran K. P., J. N. Gray, R. A. Lorie, y I. L. Traiger: "The notions of consistency and predicate locks in a database system", *Comm. ACM* 19:11, pág. 624-633. 1976.

[83] Fagin, R.: "Multivalued dependencies and a new normal form for relational databases", *ACM Trans. on Database Systems* 2:3, pág. 262-278. 1977.

[84] Fagin, R.: "On an authorization mechanism", *ACM Trans. on Database Systems* 3:3, pág. 310-319. 1978.

[85] Fagin, R.: "Normal forms y relational database operators", *ACM SIGMOD Intl. Conf. on Management of Data*. 1979.

[86] Fagin, R.: "Horn clauses and database dependencies", *J. ACM* 29:4, pág. 952-983. 1982.

[87] Fagin, R., J. Nievergelt, N. Pippenger, y H. R. Strong: "Extendible hashing -a fast access method for dynamic files", *ACM Trans. on Database Systems* 4:3, pág. 315-344. 1979.

[88] Fagin, R. y M. Y. Vardi: "The theory of data dependencies -a survey", *Mathematics of Information Processing* (M. Anshel y W. Gewirtz, eds.), *Symposia in Applied Mathematics* 34, pág. 19-72. 1986.

[89] Fekete, A., N. Lynch, M. Merritt, y W. Weihl: "Nested transactions and read/write locking", *Proc. Sixth ACM Symp. on Principles of Database Systems*, pág. 97-111. 1987.

[90] Fernández, E. B., R. C. Summers, y C. Wood: *Database Security and Integrity*, Addison-Wesley. 1980.

[91] Finkel, R. A. y J. L. Bentley [1974]: "*Quad* trees, a data structure for retrieval on composite keys", *Acta Informatica* 4:1, pág. 1-9. 1974.

[92] Fischer, P. C. y D. M. Tsou: "Whether a set of multivalued dependencies implies a join dependency is NP-hard", *SIAM J. Computing* 12:2, pág. 259-266. 1983.

[93] Fischer P. C. y D. Van Gucht: "Weak multivalued dependencies", *Proc. Third ACM Symp. on Principles of Database Systems*, pág. 266-274. 1984.

[94] Franaszek, P. y J. T. Robinson: "Limitations on concurrency in transaction processing", *ACM Trans. on Database Systems* 10:1, pág. 1-28. 1985.

[95] Fredman, M. F.: "A lower bound on the complexity of orthogonal range queries", *J. ACM* 28:4, pág. 696-705. 1981.

[96] Furtado, A. L.: "Formal aspects of the relational model", *Information Systems* 3:2, pág. 131-140. 1978.

[97] Galil, Z.: "An almost linear time algorithm for computing a dependency basis in a relational database", *J. ACM* 29:1, pág. 96-102. 1982.

[98] Gallaire, H. y J. Minker: *Logic and Databases*. Plenum Press, New York. 1978.

[99] Gallaire, H., J. Minker, y J. M. Nicolas: *Advances in Database Theory*. Plenum Press, New York. 1981.

[100] Gallaire, H., J. Minker, y J. M. Nicolas: *Advances in Database Theory*. Plenum Press, New York. 1983.

[101] Gallaire, H., J. Minker, y J. M. Nicolas: "Logic and databases: a deductive approach", *Computing Surveys* 16:1, pág. 154-185. 1984.

[102] Garcia-Molina, H. y J. Kent: "An experimental evaluation of crash re-

covery algorithms", *Proc. Fourth ACM Symp. on Principles of Database Systems*. pág. 113-121. 1985.

[103]. Garey, M. R. y D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco. 1979.

[104]. Gelenbe, E. y D. Derochette: "Performance of rollback recovery systems under intermittent failures", *Comm. ACM* 21:6, pág. 493-499. 1978.

[105]. Gelfond, M., H. Przymusinska, y T. C. Przymusinski: "The extended closed world assumption and its relationship to parallel circumscription", *Fifth ACM Symp. on Principles of Database Systems*, pág. 133-139. 1986.

[106]. Genesereth, M. R. y N. J. Nilsson: *Logical Foundatations of Artificial Intelligence*, Morgan Kaufmann, Los Altos. 1988.

[107]. Ginsberg, M.: *Nonmonotonic Reasoning*. Morgan Kaufmann, Los Altos. 1988.

[108]. Ginsburg, S. y S. M. Zaiddan: "Properties of functional dependencies families", *J. ACM* 29:3. pág. 678-698. 1982.

[109]. Gottlieb, G.: "Computing covers for embedded functional dependencies", *Proc. Sixth ACM Symp. on Principles of Database Systems*, pág. 58-69. 1987.

[110]. Graefe, G.: "Query evaluation techniques for large databases", *ACM Comp. Surveys* 25:2. 1993.

[111]. Graefe, G.: *Query Processing for Large Databases*. Morgan Kaufmann 1998.

[112]. Graham M. H., A. O. Mendelzon y M. Y. Vardi: "Notions of dependency satisfaction", *J. ACM* 33:1, pág. 105-129. 1986.

[113]. Gray, J. N.: "Notes on database operating systems" en *Operating Systems: an Advanced Course* (R. Bayer, R. M. Graham, y G. Seegmuller, eds.) Springer-Verlag, New York. 1978.

[114]. Gray, J. N., et al.: "The recovery manager of the system R database, manager", *Computing Surveys* 13:2, pág. 223-242. 1981.

[115]. Gray, J. N., R. A. Lorie, y G. R. Putzolo [1975] "Granularity of locks in a shared database", *Proc. Intl. Conf. on Very Large Data Bases*, pág. 428-451. 1975.

[116]. Gray, J. N. y A. Reuter: *Transaction Processing: concepts and techniques*. Morgan Kaufmann. 1993.

[117]. Gray, P.: *Logic, Algebra and Databases*. Wiley. 1984.

[118]. Greenblatt, D. y J. Waxman: "A study of three database query languages", en Shneiderman, pág. 77-98. 1978.

[119]. Gurevich, Y. y H. R. Lewis: "The inference problem for template dependencies", *Proc. First ACM Symp. on Principles of Database Systems*, pág. 221-229. 1982.

[120]. Hadzilacos, T. y C. H. Papadimitriou: "Some algorithmic aspects of Multiversion concurrency control", *Proc. Fourth ACM Symp. on Principles of Database Systems*, pág. 96-104. 1985.

[121]. Hadzilacos, T. y M. Yannakakis: "Deleting completed transactions", *Proc. Fifth ACM Symp. on Principles of Database Systems*, pág. 43-46. 1986.

[122]. Hadzilacos, V.: "An algorithm for minimizing roll back cost", *Proc. First Symp. on Principles of Database Systems*, pág. 93-97. 1982.

[123]. Haadzilacos, V.: "A knowledge-theoretic analysis of atomic commitment protocols", *Proc. Sixth ACM Symp. on Principles of Database Systems*, pág. 129-134. 1987.

[124]. Haerder, T. y A. Reuter: "Principles of transaction oriented database recovery - taxonomy", *Computing Surveys* 15:4, pág. 287-317. 1983.

[125]. Hagihara, K., M. Ito, K. Taniguchi, y T. Kasami: "Decision problems for multivalued dependencies in relational databases", *SIAM J. Computing* 8:2, pág. 247-264. 1979.

[126]. Hammer, M. y D. McLeod: "Database description with SDM: a semantic database model", *ACM Trans. on Database Systems* 6:3, pág. 351-386. 1981.

[127]. Held, G. y M. Stonebraker: "B-trees reexamined", *Comm. ACM* 21:2, pág. 139-143. 1978.

[128]. Honeyman, P.: "Testing satisfaction of functional dependencies", *J. ACM* 29:3, pág. 668-677. 1982.

[129]. Imielinski, T.: "Query processing in deductive database systems with incomplete information", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 268-280. 1986.

[130]. Jarke, M., J. Clifford, y Y. Vassiliou: "An optimizing Prolog front end to a relational query system", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 296-306. 1984.

[131]. Jou, J. H. y P. C. Fischer: "The complexity of recognizing 3NF rela-

tion schemes", *Information Processing Letters* 14:4, pág. 187-190. 1983.

[132] Kanellakis, P: "Elements of relational database theory", *Handbook of Theoretical Computer Science* (J. Van Leeuwen ed.), Elsevier. 1991.

[133] Kedem, Z. y A. Silberschatz: "Controlling concurrency using locking protocols", *Proc. Twentieth Annual IEEE Symp. on Foundations of Computer Science*, pág. 274-285. 1979.

[134] Kedem, Z. y A. Silberschatz: "Non-two phase locking protocols with shared and exclusive locks", *Proc. Intl. Conf. on Very Large Data Bases*, pág. 309-320. 1980.

[135] Keller, A.: "Algorithms for translating view updates into database updates for views involving selections, projections, and joins", *Proc. Fourth ACM Symp. on Principles of Database Systems*, pág. 154-163. 1985.

[136] Kim, W.: "Relational database systems". *Computing Surveys* 11:3, pág. 185-210. 1979.

[137] Kim, W. (ed.): *Modern Database Systems*. ACM Press-Addison-Wesley. 1995.

[138] Kim, W., D. S. Reiner y D. S. Batory (eds.): *Query Processing in Database Systems*. Springer-Verlag. 1985.

[139] Klug, A.: "Equivalence of relational algebra and relational calculus query languages having aggregate functions", *J. ACM* 29:3, pág. 699-717. 1981.

[140] Knuth, D. E.: *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms* (3ª ed.), Addison-Wesley. 1997.

[141] Knuth, D. E.: *The Art of Computer Programming*, Vol. 3, *Sorting and Searching* (2ª ed.), Addison-Wesley. 1998.

[142] Korth, H. F.: "Locking primitives in a database system", *J. ACM* 30:1, pág. 55-79. 1983.

[143] Korth, H. F.: "Deadlock freedom using edge locks". *ACM Trans. on Database Systems* 7:4, pág. 632-652. 1982.

[144] Kowalski, R. A. [1974]. "Predicate logic as a programming language", *Proc. 1974 IFIP Congress*, pág. 569-574, North Holland, Amsterdam. 1974.

[145] Kumar, V.: *Performance of Concurrency Control Mechanism in Centralized Database Systems*. Prentice-Hall. 1996.

[146] Larson, P.: "Dynamic hashing", *BIT* 18:2, pág. 184-201. 1978.

[147] Lehman, P. L. y S. B. Yao: "Efficient locking for concurrent operations on B-trees", *ACM Trans. on Database Systems* 6:4, pág. 650-670. 1981.

[148] Ling, W., F. W. Tompa y T. Kameda: "An improved third normal form for relational databases", *ACM Trans. on Database Systems* 6:2, pág. 329-346. 1981.

[149] Litwin, W.: "Linear hashing: a new tool for file and table addressing", *Intl. Conf. on Very Large Data Bases*, pág. 212-223. 1980.

[150] Lucchesi, C. L. y S. L. Osborn: "Candidate keys for relations", *J. Computer y System Sciences* 17:2, pág. 270-279. 1978.

[151] Maier, D.: "Minimum covers in the relational database model", *J. ACM* 7:4, pág. 664-674. 1980.

[152] Maier, D.: *The Theory of Relational Databases*. Computer Science Press, Rockville, Md. 1983.

[153] Maier, D., A. O. Mendelzon, F. Sadri, y J. D. Ullman: "Adequacy of decompositions in relational databases", *J. Computer y System Sciences* 21:3, pág. 368-379. 1980.

[154] Maier, D., A. O. Mendelzon, y Y. Sagiv: "Testing implications of data dependencies", *ACM Trans. on Database Systems* 4:4, pág. 455-469. 1979.

[155] Maier, D., Y. Sagiv, y M. Yannakakis: "On the complexity of Testing implications of functional and join dependencies", *J. ACM* 28:4, pág. 680-695. 1981.

[156] Manna, Z. y R. Waldinger: *The Logical Basis for Computer Programming*. Addison-Wesley. 1985.

[157] Mannila, H. y K. J. Raiha: *The design of relational databases*. Addison-Wesley. 1992.

[158] Melton, J. y A. Simon: *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann. 1993.

[159] Mendelzon, A. O.: "On axiomatizing multivalued dependencies in relational databases", *J. ACM* 26:1, pág. 37-44. 1979.

[160] Mendelzon, A. O. y D. Maier: "Generalized mutual dependencies and the decomposition of database relations", *Proc. Intl. Conf. on Very Large Data Bases*, pág. 75-82. 1979.

[161] Mendelzon, A. O. y T. Milo: "Formal models of web queries", *ACM SIGACT SIGMOD Symp. on Principles of Database Systems*. 1997.

[162]. Merret T. H.: *Relational Information Systems*. Reston Publishing. 1984.

[163]. Minker, J. (ed.): *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, Los Altos. 1988.

[164]. Mishra, P. y M. Eich: "Join processing in Relational Databases", *ACM Computing Surveys* 24:1, pág. 63-113. 1992.

[165]. Mittra S. S.: *Principles of Relational Database Systems*. Prentice-Hall. 1991.

[166]. Mohan, C. et al.: "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging", *ACM Trans on Database Systems* 17:1, pág. 94-162. 1992.

[167]. O'Neil P.: *Database: Principles, Programming and Performance*. Morgan Kaufmann. 1994.

[168]. O'Neil, P. y D. Quass: "Improved query performance with variants indexes", *Proc. ACM SIGMOD Conf. on the Management of Dat*, pág. 38-49. 1997.

[169]. Papadimitriou, C. H.: "The serializability of concurrent database updates", *J. ACM* 26:4, pág. 631-653. 1979.

[170]. Papadimitriou, C. H.: "Concurrency control by locking", *J. ACM* 12:2 pág. 215-226. 1983.

[171]. Papadimitriou, C. H.: *The Theory of Database Concurrency Control*. Computer Science Press, Rockville, Md. 1986.

[172]. Papadimitriou, C. H. y P. C. Kanellakis: "On concurrency control by multiple versions", *ACM Trans. on Database Systems* 9:1 pág. 89-99. 1984.

[173]. Pirotte, A.: "High level data base query languages", en Gallaire y Minker, pág. 409-436. 1978.

[174]. Ramakrishnan, R.- Gehrke, J.: *Database Management Systems* (2ª ed.). McGraw-Hill. 1999.

[175]. Ramarao, K. V. S.: "On the complexity of commit protocols", *Proc. Fourth ACM Symp. on Principles of Database Systems*, pág. 235-244. 1985.

[176]. Reiter, R.: "Towards a logical reconstruction of relational database theory", en Brodie, Mylopoulos, y Schmidt [1984], pág. 191-233. 1984.

[177]. Reiter, R.: "A sound and sometimes complete query evaluation algorithm for relational databases with null values", *J. ACM* 33:2, pág. 349-370. 1986.

[178]. Reuter, A.: "Performance analysis of recovery techniques", *ACM Trans. on Database Systems* 9:4, pág. 526-559. 1984.

[179]. Rissanen J.: "Theory of joins for relational databases: a tutorial survey", *Proc. Seventh Symp. on Mathematical Foundations of C. S.*, Lectures notes in CS, 64, pág. 537-551. Springer-Verlag. 1979.

[180]. Rivest, R. L.: "Partial match retrieval algorithms", *SIAM J. Computing* 5:1 pág. 19-50. 1976.

[181]. Robinson, J. T.: "The K-D-B tree; a search structure for large multidimensional dynamic indices", *ACM SIGMOD Intl. Conf. on Management of Data*, pág. 10-18. 1981.

[182]. Robinson, J. T.: "Order preserving linear hashing using dynamic key statistics", *Proc. Fifth ACM Symp. on Principles of Database Systems*, pág. 91-99. 1986.

[183]. Rosenberg, A. L. y L. Snyder: "Time and space optimality in B-trees", *ACM Trans. on Database Systems* 6:1, pág. 174-193. 1981.

[184]. Sadri, F. y J. D. Ullman: "Template dependencies: a large class of dependencies in relational databases and their complete axiomatization", *J. ACM* 29:2, pág. 363-372. 1981.

[185]. Sagiv, Y., C. Delobel, D. S. Parker, y R. Fagin: "An equivalence between relational database dependencies and a fragment of propositional logic", *J. ACM* 28:3, pág. 435-453. 1981.

[186]. Sagiv, Y. y S. Walecka: "Subset dependencies and a completeness result for a subclass of embedded multivalued dependencies", *J. ACM* 29:1, pág. 103-117. 1982.

[187]. Salzberg, B. J.: *File Structures*. Prentice-Hall. 1988.

[188]. Sciore, E.: "A complete axiomatization of full join dependencies", *J. ACM* 29:2, pág. 373-393. 1982.

[189]. Schmidt, J. W. y M. L. Brodie: *Relational Database Systems: Analysis and Comparison*. Springer-Verlag. 1983.

[190]. Schneiderman, B. (ed.): *Database: Improving Usability and responsiveness*. Academic Press, New York. 1978.

[191]. Silberschatz, A., H. Korth y S. Sudarshan: *Database systems concepts* (3ª ed.). McGraw-Hill. 1998.

[192]. Smith, J. M. y D. C. P. Smith: "Database abstractions: aggregation and generalization", *ACM Trans. on Database Systems* 2:2, pág. 105-133. 1977.

[193] Stonebraker, M.: "Retrospection on a database system", *ACM Database Systems* 5:2, pág. 225-240. 1980.

[194] Stonebraker, M.: "Triggers and inference in database systems", en Brodie y Mylopoulos, pág. 297-314. 1986.

[195] Stonebraker, M. y L. A. Rowe: "The design of Postgres", *ACM SIGCMOD Intl. Conf. on Management of Data*, pág. 340-355. 1986.

[196] Stonebraker, M., B. Wong, P. Kreps, y G. Held: "The design and implementation of INGRES", *ACM Trans. on Database Systems* 1:3, pág. 189-222. 1976.

[197] Tay, Y. C., N. Goodman, y R. Sun: "Locking performance in centralized databases", *ACM Trans. on Database Systems* 10:4, pág. 415-462. 1985.

[198] Teorey, T.: *Database Modeling and design* (3ª ed.). Morgan Kaufmann. 1998.

[199] Thomasian, A.: "Concurrency Control: Methods, Performance, and Analysis", *ACM Computing Surveys* 30:1, pág. 70-119. 1998.

[200] Tsichritzis, D. C. y F. H. Lochovsky: *Data Models*. Prentice-Hall. 1982.

[201] Tsou, D. M. y P. C. Fischer: "Decomposition of a relation scheme into Boyce-Codd normal form", *SIGACT News* 14:3, pág. 23-29. 1982.

[202] Ullman, J. D.: *Principles of Database and Knowledge-base Systems*, Vol. I y II. Computer Science Press, Md. 1989.

[203] Ullman, J. D. y J. Widom: *A First Course in Database Systems*. Prentice-Hall. 1997.

[204] Ullman, J. D.: "Database theory: past and future", *Proc. Sixth ACM Symp. on Principles of Database Systems*, pág. 1-10. 1987.

[205] Valduriez, P.: "Join Indices", *ACM Trans. on Database Systems* 12:2, pág. 218-246. 1987.

[206] Van Emden, M. H. y R. A. Kowalski: "The semantics of predicate logic as a programming language", *J. ACM* 23:4, pág. 733-742. 1976.

[207] Van Gelder, A. y R. W. Topor: "Safety and correct translation of relational calculus formulas", *Proc. Sixth ACM Symp. on Principles of Database Systems*, pág. 313-327. 1987.

[208] Vardi, M. Y.: "Complexity of relational queries", *Proc. Fourteenth Anual ACM Symp. on the Theory of Computing*, pág. 137-145. 1982.

[209] Vardi, M. Y.: "Inferring multivalued dependencies from functional and join dependencies", *Acta Informatica* 19:2, pág. 305-324. 1983.

[210] Vardi, M. Y.: "Querying logical databases", *Proc. Fourth ACM Symp. On Principles of Database Systems*, pág. 57-65. 1985.

[211] Wiederhold, G.: *Database Design*, McGraw-Hill, New York. 1983.

[212] Wiederhold, G.: "Views, objects, and databases", *Computer*, Dic. 1986.

[213] Wiederhold, G.: *File Organization for Database Design*. McGraw-Hill, New York. 1987.

[214] Yannakakis, M.: "Serializability by locking", *J. ACM* 31:2, pág. 227-245. 1984.

[215] Zaniolo, C. et al.: *Advanced Database Systems*. Morgan Kaufmann. 1997.

[216] Zaniolo, C. y M. A. Melkanoff: "On the design of relational database schemata", *ACM Trans. on Database Systems* 6:1, pág. 1-47. 1981.

[217] Zaniolo, C.: "A new normal form for the design of relational database schemata", *ACM Trans. on Database Systems* 7:3, pág. 489-499. 1982.

ÍNDICE

A

abd 3 5-7

abortar, 199, 215 218 219 222-224, 229

acceso 2 3 5-7 9 29 51 52, 74 76, 146 149 151, 152, 159, 161, 168, 170 171 177 178, 183, 185 204, 217 231 239, 243, 246

acción 72 85 86 198 199 218 226, 240 241, 248

acid 200, 214

activa, 232, 240

actualización inmediata 222

administrador de base de datos, 3, 6

agregación, 11, 237

aislamiento

propiedad de, 201

niveles 212-214

algebra relacional, 19 20 29 30, 40, 43 50 54 59 69 176-178

all 57 58 65-67, 95 233

alter 53 54

anomalías de actualización, 100 101, 112 121

ansi 52, 98 231 250

árbol-B, 153-157, 159, 171, 172 174 184

árbol b+, 158, 170 172

archivos 1-3, 7, 145, 146 148, 149, 176 216

array, 227, 235, 236

asociación, 11, 76

atomicidad, 200, 201, 214

atributo, 16-18, 20-24, 41 53, 54, 58-60, 71, 77-80, 82, 83, 109, 110, 114, 125, 133 139, 150, 170, 176, 179-181, 183, 184, 187, 189, 191, 192, 233-235 241, 243, 245

autorización, 6, 52

axiomas, 105-107 129-131, 138, 139, 142, 143

B

base de datos

activa, 232, 240

administrador de (véase abd)

consistencia, 203

esquema, 5 20

estado, 12
integridad, 79
recuperación, 29
usuarios, 74
bitácora, 215
blob, 232, 233, 246
bloque de disco, 2, 146, 147, 155, 159
boolean, 233
booleana, 32, 53, 42
operadores, 34
operaciones, 180
Boyce-Codd (fnbc), 121, 127

C
cache,
gestor de, 215, 224
memoria, 219, 220, 223, 224, 230
consistencia, 12, 200-203
cálculo de las bases, 131, 144
cálculo relacional, 20, 29, 40, 42, 43, 46, 47, 50, 51, 67, 177
candado, 204-206, 212, 213, 227
checkpoint, 217, 220, 222-224
clasificación, 11, 26-28
clausura, 106
de un conjunto de atributos, 107, 108, 112, 119, 124, 131, 144
de un conjunto de dependencias, 109
clave, 22, 24, 82, 86, 99, 103, 104, 108, 109, 122, 125, 126, 139, 151-162, 164-168, 171, 172, 174, 181, 188, 191, 229, 245, 248
externa, 83
primaria, 17, 23, 83, 100, 101, 152, 169, 170, 216
clob, 232, 233
cociente, 38, 184
commit, 199-201, 213-215, 217-225, 228, 230
comportamiento, 160, 213, 236, 242, 245, 246, 248
create row, 234
create table, 53, 80-86, 233-235
cubrimiento minimal, 109, 110, 123, 139, 141, 143

D
datos, 1-3, 5-7, 9, 10, 17, 20, 24, 27, 29, 43, 51, 53, 56, 65, 74, 79, 100-102, 109, 112, 127, 136, 145, 147, 150, 151, 161, 177, 182, 188, 192, 195-197, 199, 213-216, 218, 220, 223, 225, 231-233, 245, 246
definición de, 53
gestor de, 7
Modelo de, 9-13, 18, 20, 24, 231
declare, 87, 88, 244, 248
dependencia, 2, 106, 109-111, 115, 116, 122, 123, 125, 127, 128, 132, 135-137, 142
funcional, 13, 103, 104, 121, 122, 125, 127, 128
deshacer, 218-223, 230
diagramas de entidad-interrelación, 15, 17
división, 38, 67
operador, 38
durabilidad, 200, 201, 214

E
entidad, 14-17, 22-24
integridad de, 13
esquema, 3, 5, 16, 17, 20-24, 28, 47, 51-53, 65, 90, 94, 96, 97, 100-105, 107-109, 112, 114, 115, 117, 118, 121-128, 130-137, 139, 140-143, 149, 153, 159, 163, 170, 175, 178, 236, 239, 247
Definición, 6

except, 65, 67, 237
exec sql, 87, 90, 199, 248
execute, 88-90
externa, 83

F
Fetch, 87, 88, 215, 223
foránea, 83, 86, 170
forma normal, 121, 124-127, 132, 137, 142
formato, 71, 85, 87-89, 238, 239
función, 2, 6-8, 16, 29, 35, 38, 71, 73, 79, 94, 96, 151, 159-161, 164, 168, 172, 173, 188-191, 198, 238, 243-247
producto cartesiano, 35
interrelación, 11, 13-17, 23, 24

G
generalización, 11, 103, 127, 133, 137, 153
gestor, 6, 7, 147, 148, 214-216, 219, 220, 224
grafo de espera, 212
grafo de precedencia, 207, 208, 211, 226

H
hash, 160, 161, 164, 172, 173, 185, 188, 189, 196
herencia, 245, 246

I
implicación de, 104, 107, 108, 137, 147
indexación, 147, 151, 152, 168-170
índice, 151-153, 159, 170, 183-188, 191, 192, 194, 195
integridad de, 13, 79
integridad referencial, 83, 84, 85
intersect, 65, 66, 237
iso, 52, 98, 231, 250

J
join, 61-66, 95
junta, 36, 37, 41, 58, 59, 61-63, 102, 103, 117, 118, 121, 127, 136, 137, 144, 147, 148, 176, 178-181, 185-193, 196
junta natural, 20, 37, 38, 180, 185, 190

L
lectura no repetible, 213, 214
lectura sucia, 213, 214, 227, 228

M
método hash, 188
modelo entidad-interrelación, 9, 13, 14, 16, 17, 22, 23, 28, 143
multivaluadas, 127, 129-131, 133, 135, 144

N
natural join, 61
not exists, 61, 67-69, 96

O
observadores, 234, 243
operador, 27, 30, 32-35, 37, 41, 45, 54, 57, 65-67, 69, 70, 95, 102, 239
división, 38
operadores
de comparación, 57
lógicos, 78
optimización de consultas, 6, 177, 178, 196
order by, 54, 71, 169, 233
outer join, 62-64, 95

P
página, 161-164, 166-168, 215, 219, 220, 223, 230

lismo, 6, 7, 197, 199, 201, 203, 211, 230
olador de 7, 8
rolador función de 7
ia, 215
to cartesiano 18, 20, 35, 36, 38, 6, 59, 176, 179, 181
didad global 164, 167, 168
lidad local 161-164, 167
ción 20, 33, 34, 38, 40, 41, 44, 6, 74, 101, 121, 123, 125, 127, 35, 137, 176, 179, 182

ación 4, 29, 51, 197, 201, 214, 217, 219, 220, 222, 224, 225, 232, 233
a de 8
e inferencia 105, 106, 129, 130, 217, 220-223, 230
215, 219, 220, 221, 223, 224
11, 16, 18-24, 26, 28, 30-42, 53-55, 58-61, 69-78, 93, 95, 107, 112-114, 118, 121, 125, 9, 132, 133, 135, 137, 139, 13, 148, 150-152, 169, 176, 181, 4-188, 191-193, 229, 231, 237

ón
ica 13
ones
tas 12
ntes 12
199, 238
236, 241, 243

: 237, 238
d 3-5, 52, 76, 98, 217, 232, 238
20, 30, 31, 33, 36, 38-40, 54, 174, 176, 178, 179, 182, 193

select 4, 54-78, 81, 82, 87, 88, 96, 149, 168, 169, 171, 175, 182, 183, 193-196, 234, 235, 237, 239, 248
in 60
serializabilidad 201, 206, 208, 214
serializable 203-214, 226-228
sgbd 3-7, 9, 13, 29, 78, 80-82, 84, 86, 90, 96, 200, 201, 214-216, 219, 250
arquitertura de
similar, 62, 87, 131, 176, 211, 225, 234, 236, 249
sql 4, 6, 14, 29, 51-53, 55, 58, 61, 64, 67-69, 72, 76, 78-80, 83, 86-94, 96, 98, 149, 175, 182, 194, 199, 216, 217, 231, 232, 236, 242, 243, 246-250
sql dinámico, 88
sql3 98, 231
sql92, 52, 58, 67, 84, 98, 199, 213, 214, 231, 232, 247
sql:1999 52, 98, 199, 231, 232, 236, 237, 240, 242, 245, 247, 248, 250
subtipo 243, 245, 246
superclave 103, 104, 108, 113, 121, 122, 125, 133, 139

I

tabla 5, 6, 18-20, 39, 55, 57, 62, 63, 66, 71, 75, 78, 81, 83-86, 95-97, 127-129, 136, 137, 148, 161, 162, 170, 188, 189, 191, 192, 213, 215, 216, 229, 231-235, 237, 239, 241-243, 245, 249
teorema, 43, 208-211
tipo 1, 2, 4, 10-13, 20, 22-24, 48, 62, 76, 82, 84, 93, 101, 103, 113, 121, 127, 149, 161, 168-170, 183, 188, 190, 215, 216, 222, 224, 233-236, 243, 245-248
de dato 9, 20, 232-236, 242, 247
subtipos 11, 245
de modelo de datos 9

de operaciones 12
de usuario 4
tuplas 18, 19, 21, 22, 30-36, 38, 40, 43-46, 50, 53-55, 59, 61, 65-67, 69, 71, 73, 74, 77, 78, 85-87, 97, 100, 102-105, 113, 122, 126-129, 137, 142, 143, 148, 149, 169, 170, 172, 175, 176-180, 183-191, 193, 194, 196, 198, 229, 231, 240

U

unión 20, 23, 34, 35, 41, 45, 65, 66, 106, 130, 131, 164, 180
union 65, 66, 95
unique 80, 236
update 72, 73, 76, 77, 85, 86, 89, 90, 241
usuario 4, 5, 13, 26, 55, 75-77, 88, 98, 141, 151, 169, 175, 229, 233, 238-240, 242, 243

W

where 4, 54-62, 69-76, 78, 87, 89, 90, 96, 149, 168, 169, 171, 175, 182, 183, 193-196, 234, 236, 241, 248